Jos 学习笔记

wizardforcel



目錄

Jos 学习笔记

作者：ROger__wonG

来源：锐之锋芒 - JOS学习笔记

Lab 1

一、

来源：JOS学习笔记（一）

> 起初　神创造天地。
>
> 地是空虚混沌．渊面黑暗．　BIOS运行在水面上。
>
> 神说、要有mbr、就加载了mbr。
>
> 神看mbr是好的、就用mbr加载了kernel。
>
> 神称mbr为引导程序、称kernel为内核．有晚上、有早晨、这是头一日。

用课余时间重新拾起JOS，作为一个码农通过了解不同技术层面的机制对自己的技术水平提高非常大，而JOS作为一个MIT的开放课程，可让我们从一无所有构造一个自己的操作系统，这无疑是学习OS的一个非常好的方法。

然而不可否认，操作系统本身是非常复杂的，即使是一个简化过的、只有基本功能的OS，里面的代码也够我研究好久，所以我在学习之余写这么几篇博客，当作学习笔记。

我使用的是6.828版本。地址http://pdos.csail.mit.edu/6.828/2011/schedule.html

1、环境搭建

1. 建立一只ubuntu11.10虚拟机，刚做好的系统是裸系统并，没有装任何东西。
2. 装git,vim,cscope,qemu,eclipse(本来想用vim+cscope看代码的，结果因为本人太低端，vim还是玩不转，所以又装了eclipse用来看代码)

2、start

首先让我们从lab1开始，lab1的目的也就是让我们熟悉一下os的启动过程，所以这篇笔记也就不拘泥于里面的excerise了而直接尝试去理解里面的代码。

先按lab1的pdf里的说明将jos的代码git下来：

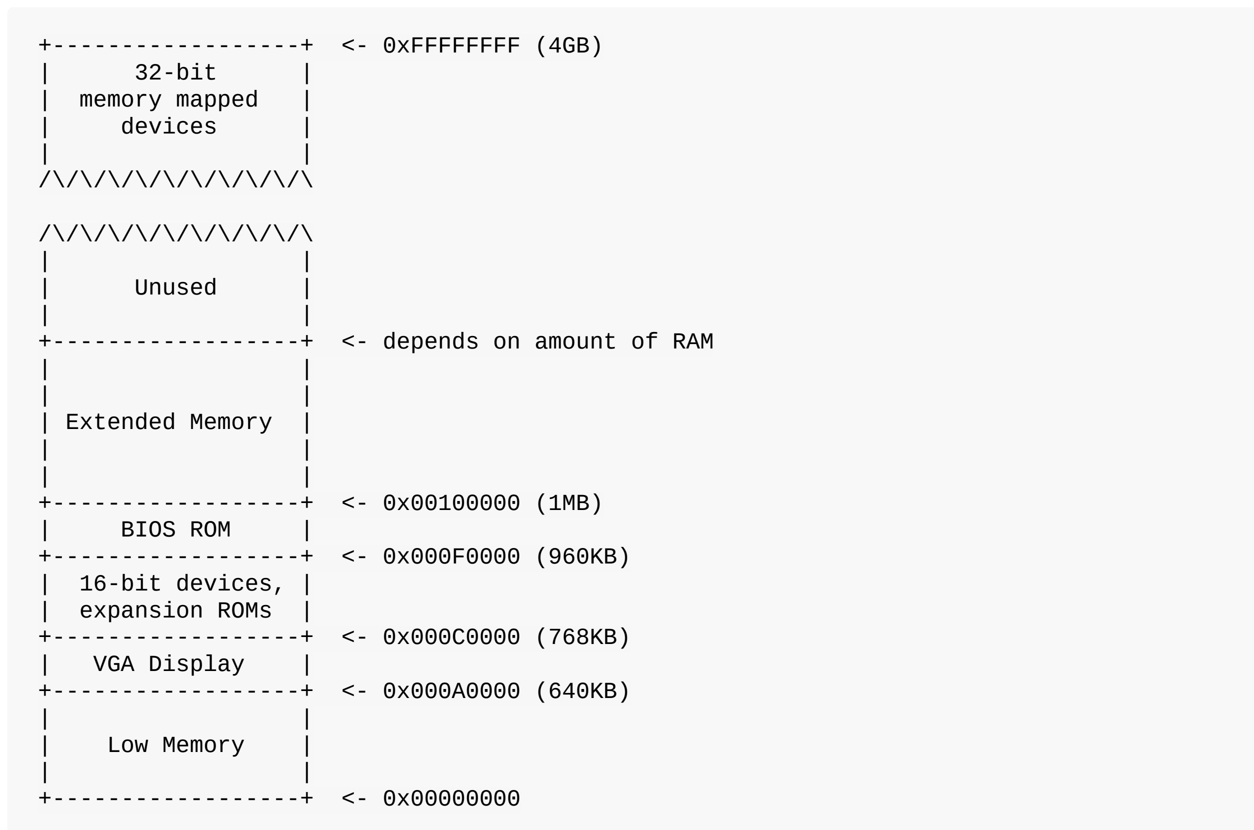
```
git clone http://pdos.csail.mit.edu/6.828/2011/jos.git lab
```

然后拖到eclipse里，我们就可以阅读代码了。

通过讲义（或者是经验）我们知道，当计算机加电，首先会把bios加载进内存执行，然后bios从硬盘加载mbr，之后由mbr来加载操作系统或者grab之类的东西。 那么这个过程我们就会面临很多问题

（1）bios加载进了内存的什么地方？

直接上图不解释：



```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

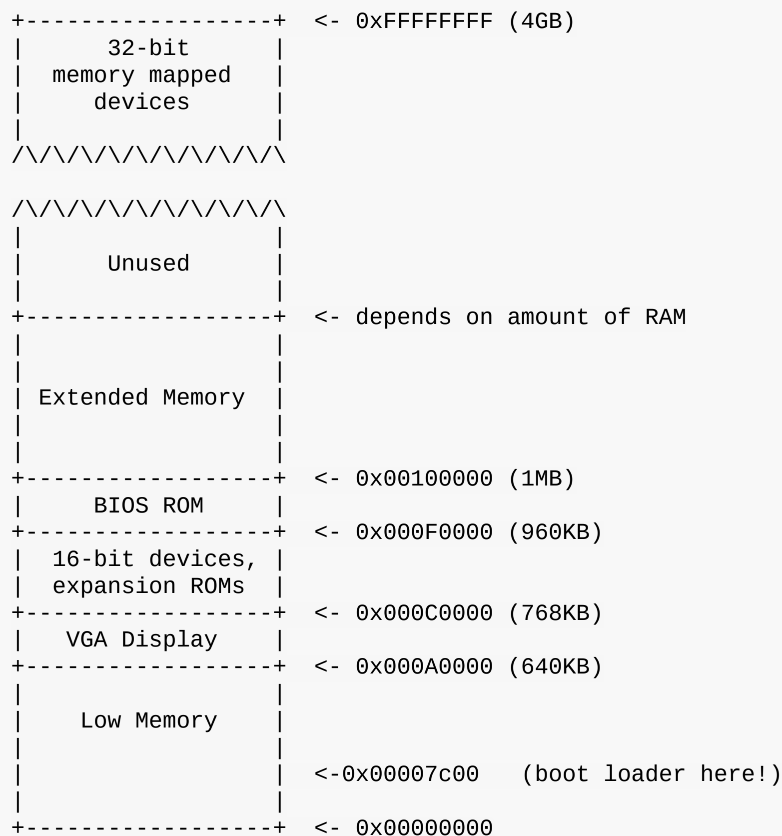
通过图我们可以看到加载到了９６０Ｋ－－１ＭＢ的地方，为啥会加载到这个位置，估计是一些约定俗成的规定吧。

（2）mbr被加载到了哪里？

在项目文件夹下make一下，编译好内核然后打开obj/boot/boot.asm，这是编译好的boot的反汇编文件，在刚开始的那几行我们就能很明显的看到有这么一个标志：

```
00007c00 <start>:
```

这说明start符号在内存中的位置“应该是”0x7c00，换句话说，这段程序“认为”它所处的位置是内存中的0x7c00，因此为了使mbr程序能正确的执行，bios会将其加载到0x7c00的位置，然后跳转到0x7c00，将控制权给它。这时候我们的内存布局是这个样子的：



```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
|                  |  <-0x00007c00   (boot loader here!)
|                  |
+------------------+  <- 0x00000000
```

（3）boot loader被加载进来做了些什么？

这就是一个较为复杂的问题，查看源码lab/boot.S，这是一个汇编文件，读起来比较痛苦，当然还是比那个asm的反汇编要强很多。

首先关中断，估计是在执行过程中不希望被打扰，然后开A20线（貌似为了和早期PC相兼容，算了，不要在意这些细节，只关注那些操作系统本身的东西就好），然后加在段表并开保护模式。

为啥要开保护模式？原因有2，首先是只有开了保护模式，才能访问64K以上的地址空间，其次是因为在实模式下程序可以访问整个地址空间的任意区域，太不安全了，因此在x86架构中引入了保护模式。

如何开启保护模式？设置cr0寄存器的某一位即可，代码：

```
movl %cr0,%eax
orl $cr0_PE_ON,%eax
movl %eax,%cr0
```

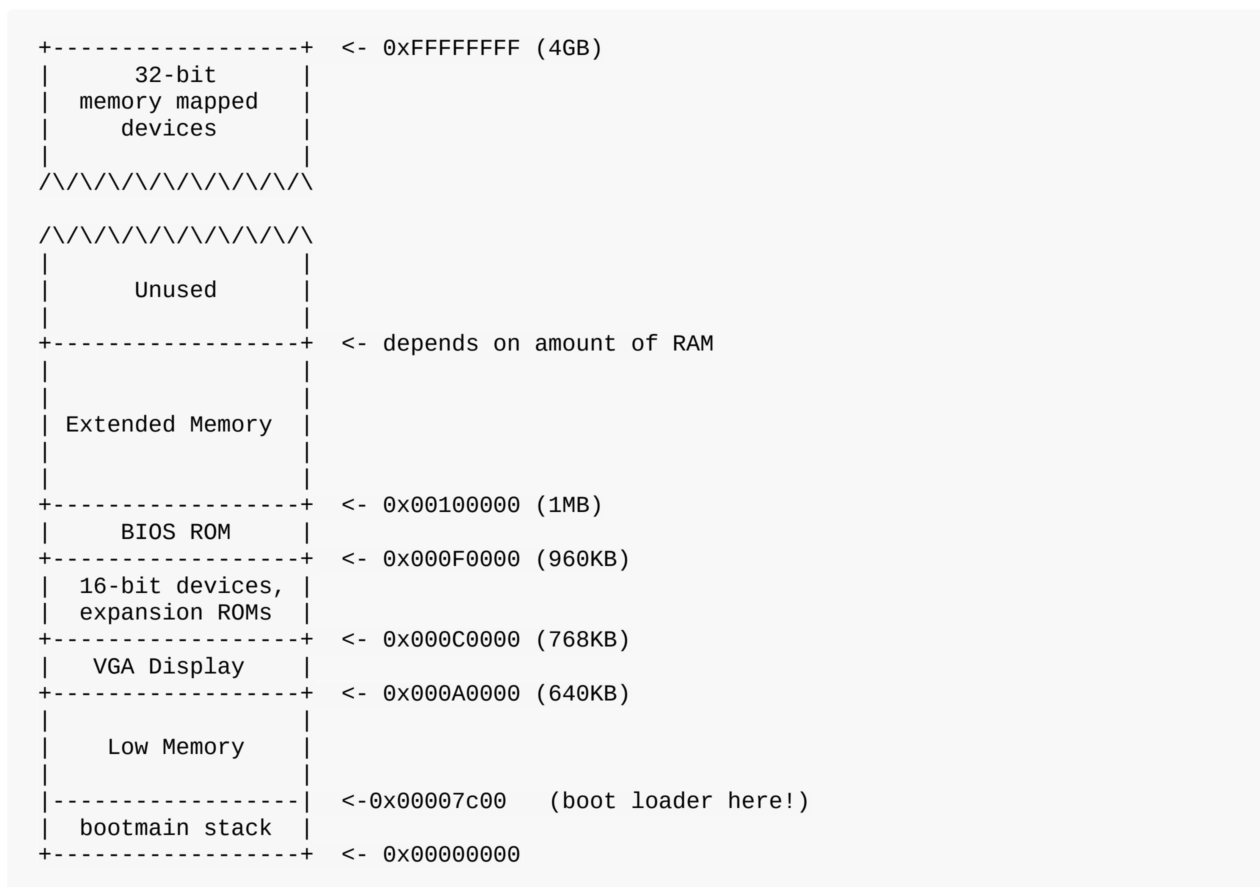
开启保护模式之后，基址：偏移这种寻址方式就变成了段选择子：偏移这种方式，而所谓的段选择子就是段表中的索引，因此为了正确的进行段式地址变换，还需要加载段表。这就是为什么在装在cr0之前需要先使用指令lgdt gdtdesc加载段表的原因。

再看段表的内容，也就是符号gdtdesc的位置，同样在boot.S这个文件下方。

可以发现gdt里面有3个段，第一个段为空段（查相关资料才知道，这是x86中的规定，第一个段均为空段），第二个和第三个段的定义使用了SEG宏，跟踪代码到mmu.h，发现宏的第一个参数是type，第二个是base，第三个是limit，所以我们可知定义的第二个和第三个段均是基质为0，长度是4G的段，也就是整个32位地址空间。

可以看出，jos并没有使用x86的段式地址变换来进行内存管理（起码在lab1里没有用），加载段表只是为了能正确的访问32位地址空间而已。

之后boot.S 设置一些寄存器的值，然后就call bootmain，跳转到boot/main.c这个文件里执行了。 值得注意的是，在任何函数调用前都要初始化栈，boot.S里很巧妙的将start作为栈的基址，因为栈空间是向下增长的，所以内存布局：



```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
|------------------|  <-0x00007c00   (boot loader here!)
|  bootmain stack  |
+------------------+  <- 0x00000000
```

(4) bootmain做了什么：

bootmain终于是c文件了，终于不用再看蛋疼的汇编文件了。

bootmain负责的事情是讲硬盘上的kernel加载到内存里并执行，那么第一个问题是，这个kernel存在硬盘的什么地方。

因为我们现在没有任何文件系统，所以jos就“很友好”的将编译好的kernel就放在mbr的后面，也就是第二个扇区（也可以说是第1个扇区，在这之前还有第0个扇区）的位置。

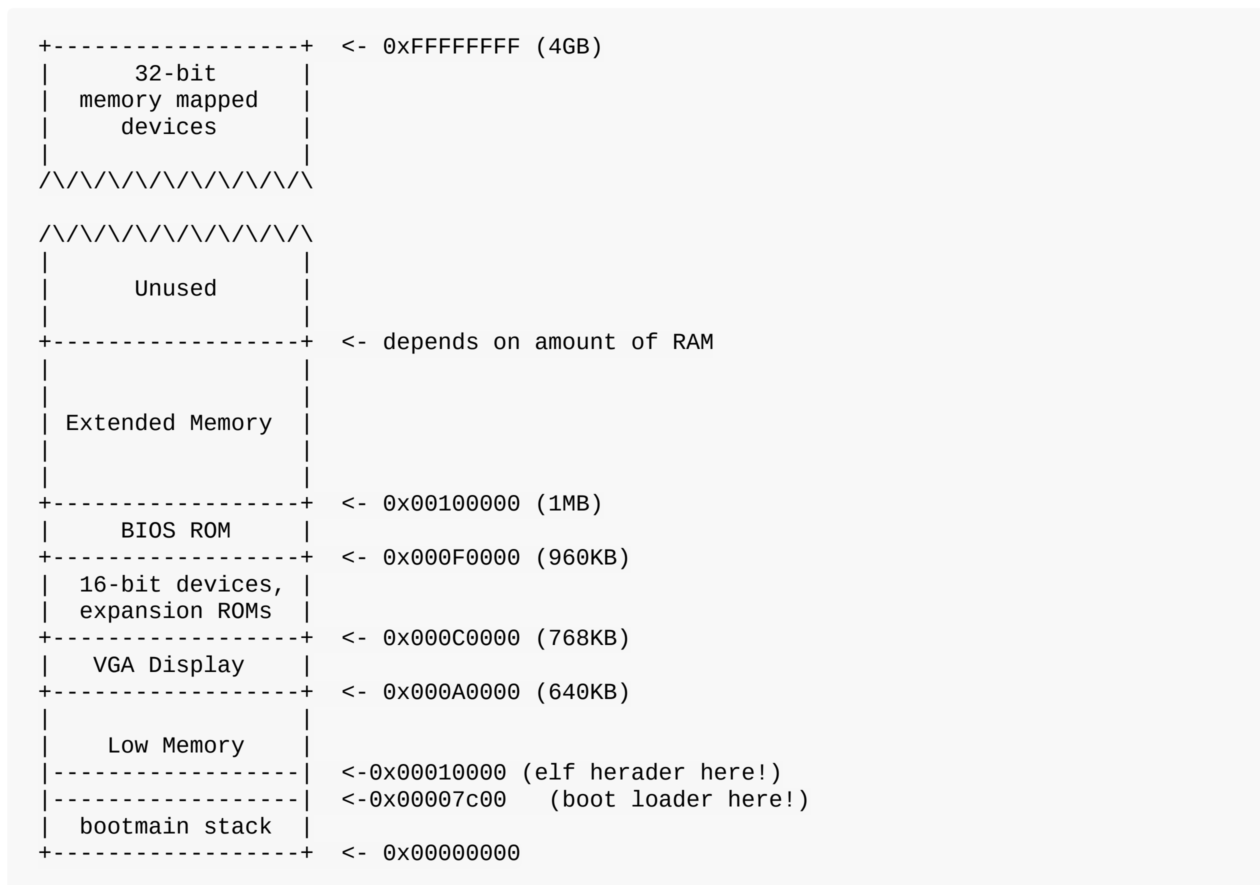
然后main.c里定义了两个函数，readseg和readsect，从逻辑上将，第一个函数的功能是“将相对于kernel基址便宜offset个自己处之后的count各字节的东西读到物理地址pa处”，而第二个的功能是“将相对于第二个扇区便宜offset字节的扇区里的内容读到dst的位置”。

第一个函数调用第二个函数完成自己的功能，第二个函数牵扯到硬盘数据的读写，当然我没有过多的花精力在这些更底层的内容上，不过看代码貌似是将地址的不同位写出到不同的端口（应该就是地址线吧），然后等待读取。

大致明白了这两个函数，就可以去看bootmain的逻辑了。

首先将8*512B的字节读到ELFHDR处，而ELFHDR是指向Elf结构体的指针，Elf结构代表一个elf头，接着通过elf头里的信息读出这个elf的其它部分，并加载到相应地址上。

ELFHDR的位置是0x10000，所以elf头会被加载到内存的这个位置（确切的说是线性地址（未经过段式变换的地址）的这个位置，但因为段表中的段基址是0，因此实际加载的位置也就是内存的位置，大概在bootloader上面一点点。



```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|------------------|  <-0x00010000 (elf herader here!)
|------------------|  <-0x00007c00   (boot loader here!)
|  bootmain stack  |
+------------------+  <- 0x00000000
```

通过elf header，并读取里面的信息，可以逐段的把elf里面的段加载进来，并加载到相应位置，至于加载到哪里，要由ELF头里面的信息所决定。因此，我们应该先研究一下elf头里读到的信息。

使用objdump -h obj/kern/kernnel可以看一下kernel的obj信息：

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00001975  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000006dc  f0101980  00101980  00002980  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00003ad5  f010205c  0010205c  0000305c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      0000199e  f0105b31  00105b31  00006b31  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         0000a300  f0108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .bss          00000660  f0112300  00112300  00013300  2**5
                  ALLOC
  6 .comment      0000002a  00000000  00000000  00013300  2**0
                  CONTENTS, READONLY
```

bootmain所做的工作就是先读到file off得到在文件中的偏移，然后再读取Size个字节，之后放到LMA所指定的地址处。

之后bootmain找出这个elf文件的entry（也在elf头里），然后跳转到这个头执行。

这样bootloader的工作就算完成了，接下来就是内核的工作了。

二、

来源：JOS学习笔记（二）

接上篇，文件跳转到了entry.S里面，这是kernel的入口。首先面临这么一个问题，kernel被加载到了什么地方？

回想上篇elf文件的加载机制，以及objdump里打印出的kernel信息，可以看到，kernel的代码段（text段）被加载到了0x100000的位置，也就是1m的位置，所以内存布局如下：

```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|------------------|
|       kernnel    |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|------------------|  <-0x00010000 (elf herader here!)
|------------------|  <-0x00007c00   (boot loader here!)
|  bootmain stack  |
+------------------+  <- 0x00000000
```

值得注意的是kernel的VMA地址为0xf0100000，也就是内核“认为”自己是在一个高位内存里执行的，因此其中的符号，包括函数名、汇编里定义的符号，都会指向一个高位的地址（大于0xf0000000)的地址，所以到目前为止在entry.S里，只要调用任何和“符号”相关的操作，比如jmp指令等，均会出错，因为高位地址连是否存在都不知道，更不用说里面是否有内容了（qemu默认是256m内存吧？）。

因此，为了使代码正常工作，需要对内存地址进行一定的映射。

上篇文章也分析了，JOS开启的8086分段机制实际上只是一个幌子，根本没有对空间进行任何的映射，所以进行映射的工作一定要由分页机制来完成。但是为了使开启分页之前的代码能正常工作，在entry.S里面定义了宏reloc。这个宏就是将一个地址减去一个kernelbase，

kernelbase定义在memlayout里面，可以看到是0xf0000000，就目前来说，任何一个高位地址减去此值，就能得到实际在物理内存中的地址了。

在定义了此宏之后，entry.S首先对entry符号和pgdir符号（也就是页表目录）地址做了重定向， 接着通过更改cr3寄存器里的某些位，开启分页内存转换。

我们跟一下pgdir里的内容，也就是entrypgdir.c里的内容，发现对于内存做了两块映射，首先是将虚拟内存的0--4M映射到物理内存的0--4M，其次是将从0xf000000之后的4M映射到物理内存的0--4M。前者的映射是为了保证低于4M的物理地址还能正确的访问（开启分页后所有的内存地址均会做变换，即使你不想让它变换），后面的映射是为了kernel能正常的工作。

关于页目录、页表不再详细说明，毕竟这是LAB2的主要内容。

之后从低地址跳转到高地址relocate处，然后初始化内核调用堆栈（调用函数都需要堆栈，所以在进入kernel的c语言代码前需要先给它初始化好堆栈，突然觉得在做用户态编程的时候不要考虑这些内存分布、栈啥的实在是太幸福了），之后就跳转到i386_init，转入c语言代码了。

为什么要从低地址跳转到relocate处？个人认为只是想验证之前开启的分页机制、加载的页表是否有错误而已，如果不跳转，直接执行貌似也可，反正call i386_init得时候也就跳转到高地址了。

另外不要被memlayout.h里面的那个内存分布所迷惑。目前进入操作系统后的调用栈位于代码的data段（就在entry.S文件的下面定义），而从objdump取得的信息来看，这个data段是加载到了物理内存的0x0010800，大概在内核代码段上面一点，虚拟内存的位置为0xf010800，也在内核代码段上面的位置，memlayout.h给的貌似是lab2以及以后的虚拟内存分布方式。

之后就跳转到c代码执行，一切都变的简单起来了。

三、

来源：JOS学习笔记（三）

过了个年，好久没碰专业内的东西了，之前做的JOS相关的东西都快忘了，还是看了前面两篇日志才想起来。

当进入内核后基本都是比较简单的代码了，我也并没有全部分析，根据讲义要求只分析了一下printf函数和堆栈的backtrace，所以这篇日志也就写这两个方面吧。

1、 printf函数。

进入kernel后从i386_init函数开始，首先做一些初始化工作，包括部分内存的清零，初始化显示器串口等（无非是判断一下地址使光标闪动正确的位置等），然后调用了cprintf，尝试讲一个10进制的数字用8进制来表示，而这个函数是需要我们完成的。

进入cprintf函数（printf.c文件）后，首先是使用va_前缀的函数（也许是宏）来取出参数，这是标准的c语言可变长度参数的实现形式。这个函数其实是一些_buildin前缀的函数的别名，而_buildin函数则是gcc内置的函数，并不在任何的JOS代码中有定义，当G C C进行编译的时候会自动将这些函数名与相应的函数体连接。之后则调用vcprintf函数。

vcprintf函数定义在同一个文件里，直接调用vprintfmt函数，值得注意的是传入一个函数指针，指向了本文件的putch函数，这个函数之后再讲。

进入vprintfmt（printfmt.c文件)函数，发现这里实现的控制打印格式的逻辑，然后调用传进来的函数指针，进行具体的打印工作。找到含有case 'o'的代码，仿照case 'd'的代码完成8进制数字显示即可，也就是原封不动的复制过来，将base改成8，如下：

```
		// (unsigned) octal
		case 'o':

			num = getint(&ap, lflag);
				if ((long long) num < 0) {
					putch('-', putdat);
					num = -(long long) num;
				}
				base = 8;
				goto number;
```

之前的putch函数会调用console.c里的cons_putc函数，而cons_putc函数又调用了cga_putc/serial_putc/lpt_putc，分别对应写显示器，写串口和并口，之所以我们不仅在qemu里面看到了kernel打印的文字，还在我们的控制台里看到了打印文字，就是因为其写了串口或者并口的原因，再由qemu将串口或并口输出信息打印到控制台（具体哪个口我没有深究）。

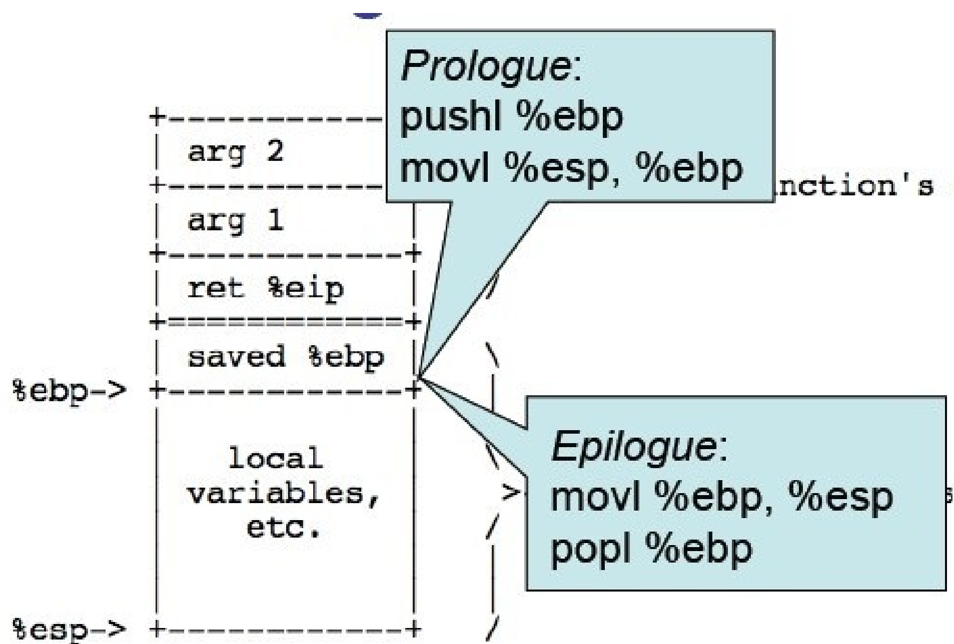
一言以蔽之，console.c完成“如何打印”的逻辑，而printfmt.c完成“打印什么”的逻辑，它们的链接纽带就是printf.c。

2、堆栈的backtrace

JOS中所有函数的调用公用一个堆栈，在之前已经分析过了堆栈是如何初始化以及位于内存中的位置，现在的任务是要从堆栈中找到函数的嵌套调用的关系，并显示出来。

首先想一想，函数调用是如何使用堆栈的。抛开JOS不谈，一个函数在调用时，肯定要压入参数给函数体传值（当然有时候也使用寄存器传值，比如windows c里面的fast call，这里暂且不论），然后要压入函数结束后的下一条指令的地址，以便函数可以正确的返回，其次因为公用一个堆栈所以要压入BP也就是基址寄存器的值，和在函数体中使用到的寄存器的值，以便返回时可以恢复现场。但是这些值压入的顺序和规则目前还是不知道的，需要一些额外的资料。

通过读kernel.asm的代码或者看讲义文件，我们可以知道JOS中的函数调用后堆栈的结构是这样的：



esp的含义是“这个地址以下的空间是未被使用的堆栈控件”，ebp的含义是“这个地址以下至esp的空间是属于目前所执行函数的堆栈空间”，所以图中saved%ebp和 ret%eip就是属于调用此函数的函数的ebp和eip。

通过阅读汇编代码我们可以发现，一个函数在调用之前，其调用者会将参数压栈（顺序没深究，和编译器有关），也就是压入arg2 和arg1，然后调用call，call的动作会把ret%eip压栈，同时转到函数体执行，在函数体执行的开头有一段预处理代码，也就是图中的prologue，会将ebp寄存器(call指令不改变ebp的值，此时的ebp还是上一个函数的）内容压栈，然后将当前esp赋值给ebp，随后进行现场保存的工作，存储在local variables空间里，值得注意的是，在预处理时会一下申请足够的空间，包括保存现场所需空间，局部变量所需空间（这大概也就

是标准C的变量声明需要放在函数开头的原因吧，为了方便编译器），调用其它函数所压入变量的空间，换句话说图中arg1,arg2是属于上一个函数的local variables空间，这也就是backtrace不能准确的判断出函数所传参数个数而统一要求打印出5个参数的原因。

因此，通过ebp不断寻找上层的ebp，直到回溯所有的函数，在entry.S中可以看到，在调用i386_init之前，将ebp置0了，因此当ebp为0的时候就是函数返回的时候，按这个逻辑代码如下：

```
int
mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{


    uint32_t eip=read_eip();
    uint32_t ebp=read_ebp();
    cprintf("Stack backtrace:");
    uint32_t esp=ebp;
    int j=0;

    while(ebp!=0)
    {
        cprintf("ebp %x eip %x ",ebp,eip);
        ebp=*(uint32_t *)(esp);
        esp+=4;
        eip=*(uint32_t *)(esp);
        esp+=4;
        cprintf("args ");
        for(j=0;j<=4;j++)
        {
            cprintf("%x ",*(uint32_t *)(esp));
            esp+=4;
        }
        cprintf("\r\n");
        esp=ebp;

    }
    return 0;
}
```

需要注意的是，因为是32位环境，所以esp指针的变化每次+4（32位是4字节）。

执行结果如下：

```
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
Stack backtrace:ebp f010ff18 eip f01008bf args 0 0 0 0 f0100944
ebp f010ff38 eip f0100087 args 0 1 f010ff78 0 f0100944
ebp f010ff58 eip f0100069 args 1 2 f010ff98 0 f0100944
ebp f010ff78 eip f0100069 args 2 3 f010ffb8 0 f0100944
ebp f010ff98 eip f0100069 args 3 4 0 0 0
ebp f010ffb8 eip f0100069 args 4 5 0 10094 10094
ebp f010ffd8 eip f0100069 args 5 1aac 660 0 0
ebp f010fff8 eip f01000ea args 111021 0 0 0 0
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

bingo!

可以看到，第一个函数参数为3个0，后面的函数参数依次是0 1 2 3 4 5，和代码逻辑相同，说明implement应该没什么大问题。

最后说一下read_eip()函数的技巧。

为什么这里read_eip()是个普通函数而不是一个inline函数，众所周知，inline函数会在编译的时候直接嵌入其调用者的代码里，这样尝试读取eip寄存器值这个动作本身就会改变eip寄存器的值，使read_eip()方法变的毫无意义。

因此作为一个普通函数，在调用的时候会把其调用者的eip入栈，然后在从栈中找到这个eip返回，栈中确切的位置之前分析过是ebp的上4个字节处，所以使用嵌入汇编高效的完成这个功能。

```
unsigned
read_eip()
{
    uint32_t callerpc;
    __asm __volatile("movl 4(%%ebp), %0" : "=r" (callerpc));
    return callerpc;
}
```

四、

来源：JOS学习笔记（四）

Lab1还差最后一部分，就是给出具体的调试信息，如下面所示：

```
K> backtrace
Stack backtrace:
  ebp f010ff78  eip f01008ae  args 00000001 f010ff8c 00000000 f0110580 00000000
         kern/monitor.c:143: monitor+106
  ebp f010ffd8  eip f0100193  args 00000000 00001aac 00000660 00000000 00000000
         kern/init.c:49: i386_init+59
  ebp f010fff8  eip f010003d  args 00000000 00000000 0000ffff 10cf9a00 0000ffff
         kern/entry.S:70: <unknown>+0
K>
```

和上次的montacktrace相比，增加了eip所在的源文件、对应的源文件行数、函数名以及在相对于该函数地址后的汇编代码的地址偏移。

首先明确，这个功能既然能实现，那么肯定在某个地方以某种方式地方记录着eip和源文件之间的相互对应的信息。

而这个存储的地方就是stab表。

首先使用objdump -G 查看一下kernel生成的obj（obj/kern/kernel)中stab表中的内容，如下：

```
5535;
79      LSYM   0      0      00000000 432    signed char:t(0,10)=r(0,10);-128;12
;
80      LSYM   0      0      00000000 470    unsigned char:t(0,11)=r(0,11);0;255
81      LSYM   0      0      00000000 507    float:t(0,12)=r(0,1);4;0;
82      LSYM   0      0      00000000 533    double:t(0,13)=r(0,1);8;0;
83      LSYM   0      0      00000000 560    long double:t(0,14)=r(0,1);12;0;
84      LSYM   0      0      00000000 593    _Decimal32:t(0,15)=r(0,1);4;0;
85      LSYM   0      0      00000000 624    _Decimal64:t(0,16)=r(0,1);8;0;
86      LSYM   0      0      00000000 655    _Decimal128:t(0,17)=r(0,1);16;0;
87      LSYM   0      0      00000000 688    void:t(0,18)=(0,18)
88      BINCL  0      0      00000000 2787   ./inc/stdio.h
89      BINCL  0      0      00000650 2801   ./inc/stdarg.h
90      LSYM   0      0      00000000 2816   va_list:t(2,1)=(2,2)=*(0,2)
91      EINCL  0      0      00000000 0
92      EINCL  0      0      00000000 0
93      BINCL  0      0      00000000 2844   ./inc/string.h
94      EXCL   0      0      00005337 720    ./inc/types.h
95      EINCL  0      0      00000000 0
96      FUN    0      0      f0100040 2859   test_backtrace:F(0,18)
97      PSYM   0      0      00000008 2882   x:p(0,1)
98      SLINE  0      13     00000000 0
99      SLINE  0      14     0000000a 0
--More--
```

前5列是stab，第六列不知道含义，第七列对应stabstr，换句话说这个表会原封不动的读入内存，stab部分的地址就是kdebug.c里的**stab_begin**，后面字符串地址就是**stabstr_begin**。

这个表很直观，根据类型不同在某些行（用index表示）会给出一个地址与源文件的对应关系（SO），紧接着给出这个源文件中的函数地址和函数名的对应关系（FUN），之后是这个函数里的每一行（SLINE）和地址偏移的对应关系。

而我们所要做的就是读取这些关系并显示出来。

问题1 这个表何时以何种方式加入的内存。

很遗憾kdebug.c里面stab表的地址符号是extern进来的，我并没有找到这个符号第一次出现的位置，不过估计应该是和内核一起，通过elf头的文件信息加载进来的，但具体加载到了哪里我没有深究。

问题2 如何对这个表进行检索。

庆幸的是，Jos很“友好”的给我们提供了一个检索函数：stab_binsearch。函数接受4个参数，第一是stab总表（这个说法并不是很恰当，虽然函数的含义是传入一个总表，但其实只要传入表中的一项当做起始位置函数也能正常工作），第二第三是表的下标的范围，代表在这个下标范围内寻找，若找到了，则第二个参数赋值为相应的表的下标，若没有找到，则right>left，第四个参数传入要寻找的表项的类型，也就是上面图的第二列。第五个参数代表要寻找的值，也就是上面图中的第五列，根据这个值寻找表项。而且为了比较高效，这个函数还是个二分搜索的。更加详细的说明函数注释都解释的很清楚。

问题3 我们需要做什么

需要做的是完成debuginfo_eip，该函数传入一个eip值，通过这个eip查找所需信息，并放入Eipdebuginfo这个结构体里。 JOS比较“友好”，已经替我们完成了很多功能了，我们只需要使用stab_binsearch找到函数地址和源文件行数对应的关系就行了。

代码如下：

```
// Search within [lline, rline] for the line number stab.
// If found, set info->eip_line to the right line number.
// If not found, return -1.
//
// Hint:
//   There's a particular stabs type used for line numbers.
//   Look at the STABS documentation and <inc/stab.h> to find
//   which one.
// Your code here.
stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
info->eip_line=stabs[lline].n_desc;
if(rline<lline)
{
    info->eip_line=-1;
}
// Search backwards from the line number for the relevant filename
```

之前的代码已经找到了函数地址，并且已经将addr减去了这个函数地址得到了函数内偏移，查找结果会放在lline里，根据这个index访问stab表即可。

之后修改backtrace函数，按要求显示相应信息即可，然后运行结果大概如下：

```
entering test_backtrace 0
Stack backtrace:ebp f010ff18 eip f01008bf args 0 0 0 0 f01009ac
kern/monitor.c :63:mon_backtrace:F(0,1)+e
ebp f010ff38 eip f0100087 args 0 1 f010ff78 0 f01009ac
kern/init.c :19:test_backtrace:F(0,18)+47
ebp f010ff58 eip f0100069 args 1 2 f010ff98 0 f01009ac
kern/init.c :16:test_backtrace:F(0,18)+29
ebp f010ff78 eip f0100069 args 2 3 f010ffb8 0 f01009ac
kern/init.c :16:test_backtrace:F(0,18)+29
ebp f010ff98 eip f0100069 args 3 4 0 0 0
kern/init.c :16:test_backtrace:F(0,18)+29
ebp f010ffb8 eip f0100069 args 4 5 0 10094 10094
kern/init.c :16:test_backtrace:F(0,18)+29
ebp f010ffd8 eip f0100069 args 5 1aac 660 0 0
kern/init.c :16:test_backtrace:F(0,18)+29
ebp f010fff8 eip f01000ea args 111021 0 0 0 0
kern/init.c :43:i386_init:F(0,18)+4d
leaving test_backtrace 0
```

值得注意的是，对应源文件，发现“行数”指向的并不是很正确，有时候会多向下指几行。

这是因为在查找stab时候，我们是根据eip进行查找，而eip的内容就是指向下一条将要执行的指令的地址，所以“行数”的不准是正常的合乎逻辑的。

当然要修正这个“bug”使之更符合我们的调试习惯也并不难，只要把查找到的lline改成lline-1，即寻找上一行源代码即可。

但因为实验貌似要求就是根据eip找到相应的位置，所以我也并没有改。

至此lab1就结束了，本来想一两周就写完的，没想到断断续续拖了好几个月。

5年前刚接触编程的时候编的第一个程序就是printf("hello,world!");，然后学习断点调试，可直到今天我才大概明白了这看上去如此“简单”而不值得一提的功能实现起来需要多少的技术水平。任何看起来“理所当然”的东西实现或者证明起来都是相当难的，数学和计算机技术都佐证了这一点。

所谓返璞归真的境界，大概就是如此吧。

Lab 2

五、

来源：JOS学习笔记（五）

> 神说、内存之间要有映射、将地址空间分为虚实。
>
> 神就造出两级页表、将变换前的地址、变换后的地址分开了．事就这样成了。
>
> 有晚上、有早晨、是第二日。

来到了lab2，内存管理，该实验分为两部分，第一部分为物理内存管理，第二部分为虚拟内存管理，本篇先描述lab1。

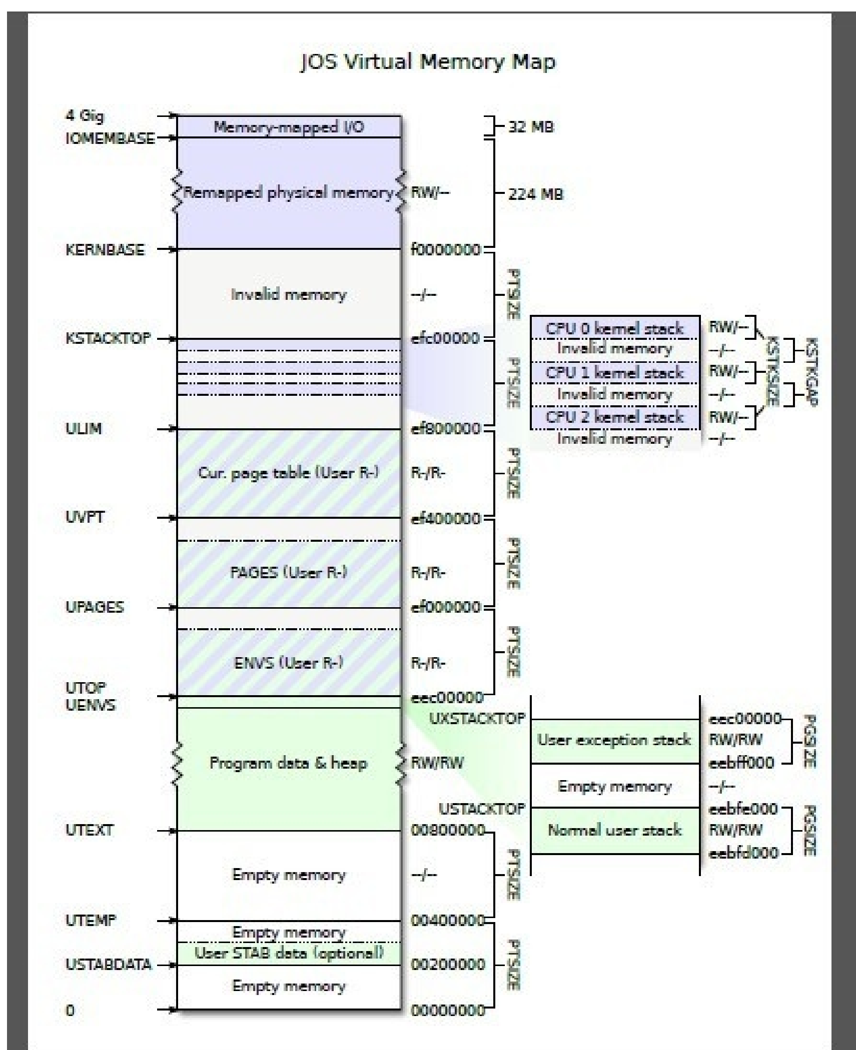
做本章实验一定头脑中要时刻清晰的记住两个内存分布图：物理内存分布图以及虚拟内存分布图。

物理内存的分布在前面的笔记中有介绍，这里拷贝过来：

```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|------------------|
|     kernnel      |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|------------------|  <-0x00010000 (elf herader here!)
|------------------|  <-0x00007c00   (boot loader here!)
|  bootmain stack  |
+------------------+  <- 0x00000000
```

虚拟内存分布参见memlayout.h文件，这里也拷贝过来：



好，记住这两个图然后开始做实验。

1、实验内容

实验内容很简单，只是让你完成下列几个函数

```
boot_alloc()
mem_init()
page_init()
page_alloc()
page_free()
```

mem_init()会调用你写的函数，然后再调用check_page_free_list和check_page_alloc两个函数来判断你完成的函数的逻辑对不对，基本上通过验证就没问题了。

2、原理。

JOS使用Page数据结构来管理内存，一个Page代表一个PGSIZE（4K）大小的物理页面，Page数据结构的定义在memlayout.h中，共有两个域，第一个域是pp_link，指向下一个空闲Page结构的指针，第二个域是一个short整形，代表当前此物理页面的引用次数，若为0则是没有被引用也就是空闲页面。

其次使用free_page_list维护一个空闲物理内存的链表，free_page_list本身就是一个Page指针，然后通过Page结构体里面的pp_link域构成空闲链表。

再次一个Page代表4K，在pmap.c中的i386_memory_detect函数中检测内存后，使用总物理内存/4k得到所需要的Page数量，赋值给npages，换句话说npages代表所需Page结构体的数量。

最后所有Page在内存（物理内存）中的存放是连续的，存放于pages处，可以通过数组的形式访问各个Page，而pages紧接于end[]符号之上，end符号是编译器导出符号，其值约为kernel的bss段在内存（虚拟内存）中的地址+bss段的段长，对应物理和虚拟内存布局也就是在kernel向上的紧接着的高地址部分连续分布着pages数组。

除此之外JOS提供page2pa,pa2page等函数可以进行Page数据结构的指针向物理地址的转换，或反转换等。

3、具体函数实现

首先被调用的是boot_alloc()，它要在什么都没做好的情况下开辟出n字节的空闲空间，并返回其首地址。

做法很简单，end符号向上均为未使用空间，只要返回这些空间就行。

首先将end符号向上和4K字节对齐（JOS已经帮我们完成），然后将这个地址（也就是nextfree）加上你要分配的空间并依然4K字节对齐，接着返回原先的nextfree即可。

```
char* result;
result=nextfree;
nextfree+=ROUNDUP(n,PGSIZE);
return result;
```

然后完成mem_init()中的部分代码，可以看到mem_init()中首先检查可用内存大小，然后调用boot_alloc()分配了一个页面给kern_pgdir，这个在part2中会用，现在没啥用。

接下来需要我们给pages分配空间。通过以上原理分析，很容易得出代码：

```
pages=(struct Page*)boot_alloc(npages*sizeof(struct Page);
```

接着完成page_init()。

在page_init()里系统首先给我们初始化了pages数组以及page_free_list，可以看到这个page_free_list指向了所有的Page结构，换句话说此时认为所有的页面都是空闲可分配的，这当然是不对的，所以就要从中把一些我们已经用的内存页面从中剔除出去，这包括0地址向上的第一个页面（包括IDT等），IO hole（0xA0000--0x100000，包括vga display ,bios等），kernel地址之上的部分（kernel本身+kern_pgdir+pages）。巧合的是，IO hole，和kernel之上部分是连续的地址，因为kernel就加载在0x100000处，所以其实只需要剔除两块地址，第一块是0地址开始的第一个页面，第二块就是io hole开始的向上的一组连续的页面。

分析一下page_free_list的代码逻辑，不难发现这个链表是从pages数组的末尾开始从高地址指向低地址，所以我们先计算出要剔除的Page的地址，然后通过指针操作剔除即可。

```
//first page
extern char end[];
pages[1].pp_link=0;
//io hole and kernel and kern_pgdir and pages
struct Page* pgstart=pa2page((physaddr_t)IOPHYSMEM);
struct Page* pgend=pa2page((physaddr_t)(end-KERNBASE+PGSIZE+npages*sizeof(struct Page)));
//cprintf("pgstart %x ,pgend %x \r\n",(int)pgstart,(int)pgend);
pgend=pgend+1;
pgstart=pgstart-1;
cprintf("pgstart %x ,pgend %x \r\n",(int)pgstart,(int)pgend);
pgend->pp_link=pgstart;
```

首先剔除第一个页面，只需让第二个页面（下标为1）的pp_link域指向空，因为原本其指向的是第一个页面，而第一个页面的pp_link域指向空，也等价与
pages[1].pp_link=pages[0].pp_link

其次剔除一组连续页面，使用pgstart和pgend代表这组连续地址空间的首尾所在的Page结构（所谓首是低地址，所谓尾是高地址）。

首地址也就是IOPHYSMEM所在地址，注意IOPHYSMEM已经是物理地址了，所以只需要使用pa2page得到Page结构即可。

pgend是较高位的地址，首先将end符号地址转化成物理地址(-KERNBASE)，然后再加上刚才分配的kern_pgdir（一个PGSIZE）和pages数组所占用的空间即可。当然更严谨点应该4K字节对齐的，不过不对其也能落在正确的4K范围内，不影响程序正确性。

其次注意到pgstart和pgend这两个Page也是要剔除的，所以需要找到pgend的上一个Page，和pgstart的下一个Page，这两个Page应该是空闲的。因为所有Page的组织是按数组进行组织，所以只需要进行+1和-1的地址操作即可。

接着改变pp_link域，跳过中间的区域即可。

接下来是很简单的page_alloc()。

代码逻辑很简单，从page_free_list头剔除一个Page，然后改变page_free_list使其为其pp_link即可。

```
struct Page *
page_alloc(int alloc_flags)
{
    cprintf("page_alloc\r\n");
    if(page_free_list==NULL)
    {
        return NULL;
    }
    struct Page* result=page_free_list;
    page_free_list=page_free_list->pp_link;
    if(alloc_flags & ALLOC_ZERO)
    {
    memset(page2kva(result),0,PGSIZE);
    }
    return result;
}
```

接着是Page_free，直接上图：

```
//
void
page_free(struct Page *pp)
{
    cprintf("page_frees\r\n");
    pp->pp_link=page_free_list;
    page_free_list=pp;
}
```

至此part1结束：

```
page_alloc
page_frees
page_frees
page_frees
check_page_alloc() succeeded!
check_pagepage_alloc
page_alloc
page_alloc
page_alloc
kernel panic at kern/pmap.c:696: assertion faile
0x0, PTE_W) < 0
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

一点感想：

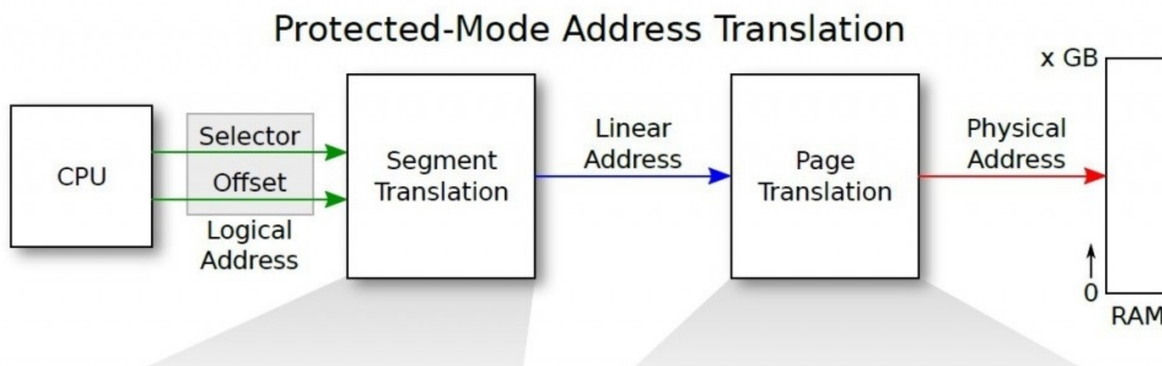
在用户态编程反而觉得内存的分配是理所当然的，然而在OS内核中写相关代码时，产生的一种很奇妙的感觉就是这个过程是由程序员自己掌控的，并且分配的结果会反而影响后面的代码逻辑。

What an amazing experience !

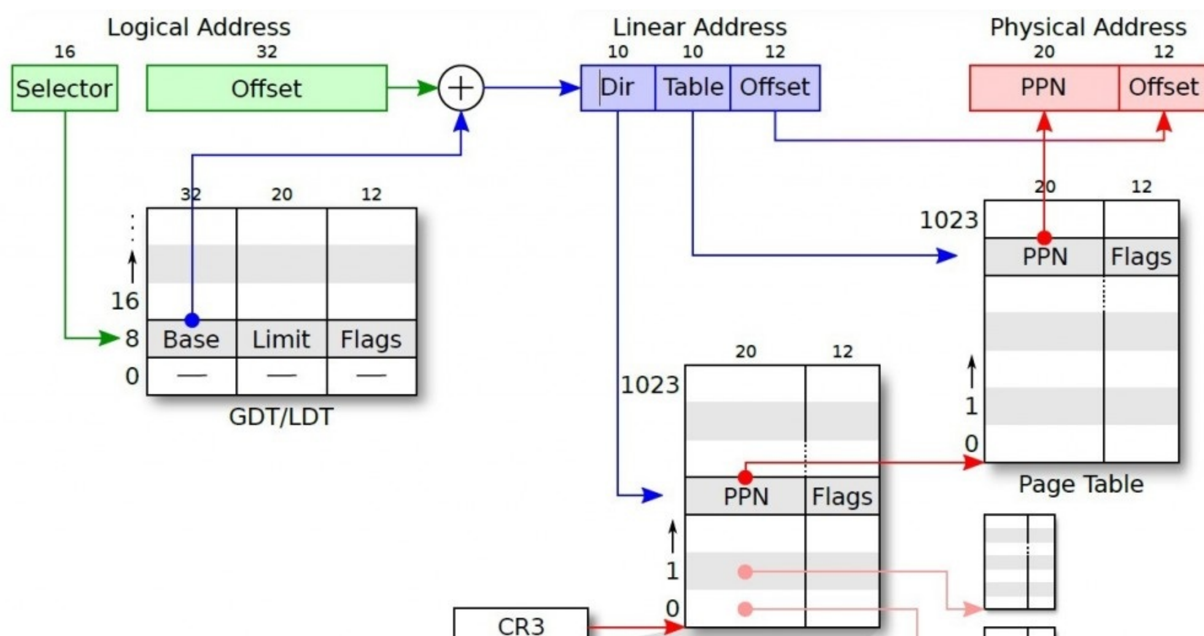
六、

来源：JOS学习笔记（六）

接下来做part2，先上一张开启分页后的地址变换图：（完整的图在 http://pdos.csail.mit.edu/6.828/2011/lec/x86_translation_and_registers.pdf ）



然后再放一张具体的地址变换的图：



好当我们把这两张图也牢记于心的时候就可以开始实验的part2了。

1、实验要求

完成以下几个函数：

```
        pgdir_walk()
        boot_map_region()
        page_lookup()
        page_remove()
        page_insert()
```

然后通过mem_init()里面的check_page函数就算过关了。

虽然要求比较简单，但实现起来可真不容易。

2、原理

（1）地址变换

首先从硬件机制说起，当cpu拿到一个地址并根据这个地址访问主存时，在x86体系架构下要经过至少两级的地址变换，第一级成为段式地址变换而第二级成为页式地址变换。（为什么？主要是从安全、兼容老的os、考虑现代os等原因）

最原始的地址叫做虚拟地址，根据规定，将前16位作为段选择子，后32位作为偏移。根据段选择子查找gdt/ldt，查到的内容替加上偏移，此时的地址就变成了线性地址。

线性地址前10位被称作页目录入口（page directory entry也就是pde），其含义为该地址在页目录中的索引，中间10位为页表入口（page table entry，也就是pte），代表在页表中的索引，最后12位是偏移。

当一个线性地址进入页式地址变换机制时，首先cpu从cr3寄存器里得到页目录（page directory）在主存中的地址，然后根据这个地址加上pde得到该地址在页目录中对应的项。无论是页目录的项还是页表的项均是32位，前20位为地址，后12位为标志位。当获取了相应的页目录项之后，根据前20位地址得到页表所在地址，加上偏移pte得到页表项，取出前20位加上线性地址本身的后12位组成物理地址，整个变换过程结束。这也就是以上两个图所描述的功能。

值得注意的是，这个过程完全是由硬件实现，在这个部分的实验中要做的是初始化并维护页目录与页表，当页目录与页表维护好了，然后使cr3装载新的页目录，一切就交由硬件去处理地址变换了。

这里还有两个点需要说下：

1、为什么是20位就能表示页表所在地址？因为页表的分配以页为单位，换句话分配的过程是分配一个页面，而这个页面所有内容都用来当做页表，而页正好是4k，所以页表地址必定是4k对齐。

2、一个页表项对应一页（也就是4K内容），因为其后面有12位的偏移。一个页表有1024个页表项，因为一个页表大小为4K，一个页表项为32位（4B）。一个页目录项对应一个页表，对应4K*1024=4M空间的映射。一个页目录有1024个页目录项，对应4M*1024=4G地址的映

射，正好是32位地址空间。

3、一个理发师只为不自己理发的人理发，当然我们在这里不套路罗素悖论。段页地址变换只为虚拟地址进行地址变换，而不为它本身的地址进行变换。换句话说变换过程中所出现的任何地址都是物理地址，在编写代码的时候尤其要注意这一点。

（2）JOS的相关部分

之前的日志里已经说过，JOS的机制中，虽然使用段式地址变换，但和没使用完全一样，因为只定义了一个段，其长度为4G，换句话说，经过段式地址变换后的内容和之前完全一样，虚拟地址和线性地址完全一样。

在part2中也就是mem_init()执行环境里，JOS已经开启了页式地址变换，但变换的比较粗糙，只是简单的把0--4M物理地址分别映射到了0--4M物理地址和0xf0000000开始的4M 地址处，之前也已经详细说过这个问题。但同时也感谢一下这种“粗糙”的变换方式，让我们在编程填充页表和页目录的时候方便了许多。（为什么？因为要往里填充物理地址，存在一个把当前符号地址转化成物理地址的过程，因为变换的粗糙，所以这个过程相对简单）。

值得注意的是，在代码里我们的所有地址均为虚拟地址，不同通过任何方法直接得到某个符号实际存在于物理内存中的地址，只有通过算才能得到我们需要的物理地址。

3、实现

（1）基本数据类型与函数说明

- pde_t 代表一个页目录项
- pte_t 代表一个页表项
- pgdir_walk() 用于查找某个虚拟地址是否有页表项，如果没有也可以通过此函数创建，值得注意的是，有页表项并不代表已经被映射。
- boot_map_region() 映射一个虚拟地址区间到一个物理地址区间，貌似在本部分没用到。
- page_insert() 将一个虚拟地址映射到一个Page数据结构，也就是映射到某个物理地址。
- page_lookup() 查找一个虚拟地址对应的Page数据结构，若没有映射返回空。
- page_remove() 解除某个虚拟地址的映射。

（2）具体实现

```
pte_t *
pgdir_walk(pde_t *pgdir, const void *va, int create)
{
    cprintf("pgdir_walk\r\n");
    // Fill this function in
    pte_t* result=NULL;
    if(pgdir[PDX(va)]==(pte_t)NULL)
    {
        if(create==0)
        {
            return NULL;
        }
        else
        {
            struct Page* page=page_alloc(1);
            if(page==NULL)
            {
                return NULL;
            }
            page->pp_ref++;
            pgdir[PDX(va)]=page2pa(page)|PTE_P|PTE_W|PTE_U;
            result=page2kva(page);
        }
    }
    else
    {
        //cprintf("%u ",PGNUM(PTE_ADDR(pgdir[PDX(va)])));
        result=page2kva(pa2page(PTE_ADDR(pgdir[PDX(va)])));
    }
        return &result[PTX(va)];


}
```

思路：

首先明确一点，返回地址是虚拟地址，要不然没有任何意义（在程序中拿到物理地址也没法用）。

查找页目录表，根据宏PDX取得页目录项（相关宏定义在mmu.h中），如果不为空，取出该项内容的前20位（PTE_ADDR宏），这是物理地址，通过此物理地址查找对应的Page结构（pa2page宏），然后获得此Page的虚拟地址（page2kva宏）。

此时的地址为页表的虚拟地址，根据偏移得到页目录项，在返回此页目录项地址。

如果前20位不为空，检查create，如果为0，返回null。否则新分配一个Page作为页表，然后自增Page 的引用，让该页目录项的前20位为页表物理地址（page2pa得到物理地址），并设置一些权限符号（不加通不过最后的检测函数），在通过此页表的虚拟地址得到相应页表项的虚拟地址并返回。

boot_map_region暂时略过，等到用的时候再说。

```
struct Page *
page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
    cprintf("page_lookup\r\n");
    // Fill this function in
    pte_t* pte=pgdir_walk(pgdir,va,0);
    if(pte==NULL)
    {
        return NULL;
    }
    if(pte_store!=0)
    {
        *pte_store=pte;
    }

    if(pte[0] !=(pte_t)NULL)
    {
        //cprintf("%x \r\n",pte[PTX(va)]);
        return pa2page(PTE_ADDR(pte[0]));

    }
    else
    {
        return NULL;
    }
}
```

首先使用pgdir_walk查找pte，如果为空则说明没有映射，返回NULL。

否则根据pte_store是否为0，先把此pte的地址存到pte_store里。

接着看这个页表项是否为0（更正确的写法应该是pte[0] & PTE_U，也就是查找PTE_U这一位是否为0，但貌似我这么写也没出错），如果为0，说明地址没有被映射（有页表项不代表被映射），返回NULL。

否则返回这个页表项的前20位所组成的物理地址所对应的Page结构。

```
void
page_remove(pde_t *pgdir, void *va)
{
    cprintf("page_remove\r\n");
    pte_t* pte=0;
    struct Page* page=page_lookup(pgdir,va,&pte);
    if(page!=NULL)
    {
        page_decref(page);
    }

    pte[0]=0;
    tlb_invalidate(pgdir,va);
}
```

这个函数逻辑很简单，首先查找此va对应的物理页面，如果此页面不为空，说明va已经映射到了物理页面，减少这个物理页面的引用次数（次数为0就释放这个页面了，相关逻辑封装在了page_decref中），然后置相应的页表项为0，并通知tlb失效。tlb是个高速缓存，用来缓存查找记录增加查找速度。

```
int
page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm)
{
    cprintf("page_insert\r\n");
    // Fill this function in
    pte_t* pte;
    struct Page* pg=page_lookup(pgdir,va,NULL);
    if(pg==pp)
    {
        pte=pgdir_walk(pgdir,va,1);
        pte[0]=page2pa(pp)|perm|PTE_P;
            return 0;
    }
    else if(pg!=NULL )
    {
        page_remove(pgdir,va);
    }
    pte=pgdir_walk(pgdir,va,1);
    if(pte==NULL)
    {
        return -E_NO_MEM;
    }
    pte[0]=page2pa(pp)|perm|PTE_P;
    pp->pp_ref++;
    return 0;
}
```

首先查找该va是否已经映射到了某个物理页面，如果映射到了，则解除映射。

使用pgdir_walk查询该地址的pte（若不存在则建立，第三个参数传入1），如果pte为空，说明没有额外的空间分配页表，因此返回-E_NO_MEM。

否则将该pte的内容填充成相应物理页面的物理地址，增加这个物理页面的引用次数。

存在一个问题，若此时的va已经映射到了物理页面，而这个页面恰好又是函数传入的pp，则这段代码不能正常的工作。

假设这个物理页面恰好是pp，则会调用page_remove尝试释放这个pp，又假设这个pp的引用次数正好又是1，则这个pp就会被释放掉，进入空闲页面的链表中。

但问题是这个pp不是空闲页面，虽然之后又为其建立了映射，但将Page结构从空闲链表中取出来的逻辑仅仅包含在page_alloc函数中，因此我们又不能简单的将其取出来，这要就会导致一个非空闲的页面（其引用是1）却出现在了空闲页面的链表中。

虽然实验中再三要求不使用特例的方式进行实现，但我实在找不到更美的方式（把remove逻辑或者page_alloc部分逻辑拿出来在这个函数里重新实现一次诸如此类的方法），所以我还是用了特例进行实现，若发现引用了同一个页面，就直接改pte即可。

基本到此这部分实验就结束了，运行后能通过：

```
pgdir_walk
6pgdir_walk
pgdir_walk
7check_page() succeeded!
kernel panic at kern/pmap.c:700: assertion failed: check_va2pa(pgdir, UPAGES + i
) == PADDR(pages) + i
```

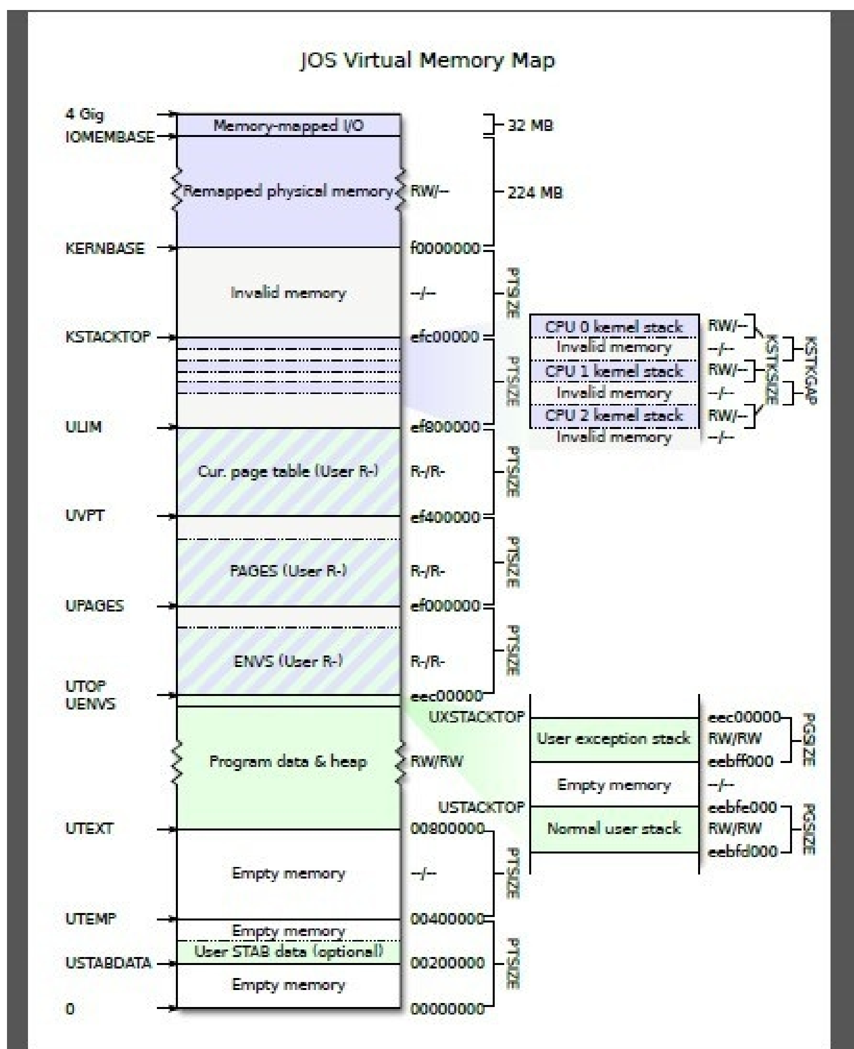
七、

来源：JOS学习笔记（七）

接前一篇。

上篇日志主要是完成了一些分页相关机制的工作，但还没有真正的去使用这个分页系统。Lab2的part3部分主要就是让我们使用part2中完成的映射机制来初始化内核的页目录和页表，并将此页目录加载到cr3里，让os真正去使用我们初始化之后的页目录以取代kernpgdir.c里面简单的页目录。

在开始之前让我们看一下JOS的虚拟内存分布图，在part3里的所有工作就是照着此图实现其中的部分映射。



首先从高位地址说起，kernbase到最高为4g是一块remapped内存，这块很大的内存从低到高要映射整块物理内存。

其次kernbase往下PTSIZE（貌似是4M）是无用内存，再往下是KERNSTACKTOP也就是内核栈的栈顶，众所周知栈的生长顺序是从高地址向低地址生长，所以这个位置也就是栈开始生长的位置。从KERNSTACKTOP往下KSTKSIZE为内核栈区域，大概是8个页面，内核栈不能超过这个区域。从KERNSTACKTOP往下一个PTSIZE这块区域除了内核栈之外还有一块空白区域，防止栈溢出的时候复写用户空间的数据。

接着是ULIM，代表用户空间的最高地址，换句话说此位置往下所有空间为用户空间，用户空间有很多东西，之后的LAB会详细说明，在此只说UPAGES。

UPAGES是JOS用户记录物理页面使用情况的数据结构，为了使用户空间能访问这块数据结构，会将PAGES映射到UPAGES位置。

好总结一下，是要总共要求映射3块区域，，1是映射UPAGES结构，，2是映射内核栈，3映射整块物理内存。

1、UPAGES的映射

所谓映射就是将某个虚拟内存地址（符号地址）映射到实际的物理地址，UPAGES映射到pages也就是通过UPAGES访问到的地址恰恰是pages这个符号指向的物理地址的内容。而映射的实质就是往页表和页目录里写入相应的值，使UPAGES这个符号经过地址变换之后指向的地址恰好是pages的物理地址。

实现的方式很简单，先获取pages数组的大小，并向上对齐到PGSIZE得到要映射的页面，然后一个页面一个页面分别映射即可。

容易出错的地方：（1）不能使用sizeof获取pages数组的大小，因为此数组是动态分配的，使用sizeof只会获取到pages这个指针本身的大小（4B）。（2）一个页面地址映射一次即可，映射的最小单位是页面，一个页面内的地址不需要反复映射。

```
//////////////////////////////////////////////////////////////////////
// Now we set up virtual memory

//////////////////////////////////////////////////////////////////////
// Map 'pages' read-only by the user at linear address UPAGES
// Permissions:
//    - the new image at UPAGES -- kernel R, user R
//      (ie. perm = PTE_U | PTE_P)
//    - pages itself -- kernel RW, user NONE
// Your code goes here:
for(i=0;i<ROUNDUP(npages*sizeof(struct Page),PGSIZE);i+=PGSIZE)
{

    page_insert(kern_pgdir, pa2page(PADDR(pages)+i),(void*) (UPAGES+i), PTE_U);

}
```

2、内核栈的映射。

目前内核栈的符号地址是bootstack表示栈底（栈的低地址，不是逻辑上的栈底反而是逻辑上的栈顶，因为向下生长），需要将KSTACKTOP-KSTKSIZE映射到这个位置，并向上按顺序映射KSTKSIZE大小的内存区域。

其余部分不用管，虽然函数说明中写了很多。

```
//////////////////////////////////////////////////////////////////////
// Use the physical memory that 'bootstack' refers to as the kernel
// stack.  The kernel stack grows down from virtual address KSTACKTOP.
// We consider the entire range from [KSTACKTOP-PTSIZE, KSTACKTOP)
// to be the kernel stack, but break this into two pieces:
//     * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by physical memory
//     * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not backed; so if
//       the kernel overflows its stack, it will fault rather than
//       overwrite memory.  Known as a "guard page".
//     Permissions: kernel RW, user NONE
// Your code goes here:
//cprintf("%x,%x \r\n",bootstack,bootstacktop);
for(i=0;i<KSTKSIZE;i+=PGSIZE)
{
    page_insert(kern_pgdir,pa2page(PADDR(bootstack)+i),(void*)(KSTACKTOP-KSTKSIZE+i),PTE_W);
}
```

3、映射整块内存区域。

要映射KERNBASE到0xffffffff总共256M内存的空间到从0开始的物理地址。虽然我们没有足够的物理空间，但we just set up the mapping anyway。什么叫just set up the mapping anyway啊，没有就是没有，不够就是不够，你让我映射到哪里去？！抱怨归抱怨，但我都给映射到了物理的0地址上去了，也算是没有办法的办法。

但这么映射整块物理内存会出问题。在page_insert操作中会将该Page结构的pp_ref++，但映射整块物理内存的过程实际是便利了整个pages数组，这就会导致所有的Page结构的pp_ref++。但别忘了还有个page_free_list串起来了空闲的Page，这些空闲的Page理所应当的pp_ref也被增加了，最后的结果是page_free_list里面页面的pp_ref不为0。在空闲链表中的页面的引用次数不为0，这很明显是不符合逻辑的，因此要进行--操作。

当然还有一种解释是：不这么做通不过check，至于愿意相信哪种随你好了。

通过这个问题我觉得这算是JOS设计不合理的一个地方，映射和页面被引用是否是一个概念？谁负责维护pp_ref的信息，是这个页面的申请者还是page_insert函数还是page_insert的调用者？感觉这些地方还有待商榷。但之前1和2的映射不用考虑这个问题是因为牵扯到的页面不在free_page_list里，这在free_page_list初始化的时候就把里面的页面剔除了。

```
//////////////////////////////////////////////////////////////////////
// Map all of physical memory at KERNBASE.
// Ie.  the VA range [KERNBASE, 2^32) should map to
//      the PA range [0, 2^32 - KERNBASE)
// We might not have 2^32 - KERNBASE bytes of physical memory, but
// we just set up the mapping anyway.
// Permissions: kernel RW, user NONE
// Your code goes here:
for(i=0;i<0xffffffff-KERNBASE;i+=PGSIZE)
{
    if(i<npages*PGSIZE)
    {
        page_insert(kern_pgdir,pa2page(i),(void*)(KERNBASE+i),PTE_W);
        pa2page(i)->pp_ref--;
    }
    else
    {
        page_insert(kern_pgdir,pa2page(0),(void*)(KERNBASE+i),PTE_W);
        pa2page(0)->pp_ref--;
    }
}
```

最后，可以看到通过check。

```
0
3
check_kern_pgdir() succeeded!
0
check_page_free_listboot_alloc
page_remove
page_remove
check_page_installed_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

lab2到此结束，lab2和lab1的代码我上传到自己的资源里欢迎下载。

Lab 3

八、

来源：JOS学习笔记（八）

> 神说、内核要有自己的数据、使用户不可访问．事就这样成了。
>
> 神称高地址为内核空间、称低地址为用户空间．　神看着是好的。
>
> 神说、用户要有自己的进程、和自己的页表、并可以进行系统调用．事就这样成了。
>
> 有晚上、有早晨、是第三日。

1、lab3概述

lab3大体分为两部分，第一部分包括执行环境（可以简单的理解为进程，下文也用进程代替执行环境）的建立和运行第一个进程，第二个部分初始化并完成中断和异常以及系统调用相关机制，本文只描述第一部分的做法。

2、原理

在建立进程之前首先要建立进程的管理机制，包括分配进程描述符的空间、建立空闲进程列表、把进程描述符数组相应内存进行映射等。这是进程描述符管理的一些工作。

其次初始化进程的虚拟地址空间，也就是给其页目录赋值过程，主要是将内核空间映射到虚拟地址的高位上。

然后加载进程的代码，代码以二进制形式和内核编译到一起了，所谓“加载”就是将这段代码从内存（内核区往上一点的位置）复制到某个物理位置，接着将这个位置和相应虚拟地址进行映射。

最后将进程的页目录加载进cr3，然后将各寄存器压栈，通过iret跳到用户态执行。

大体上讲就是这样。

3、具体实现与代码

（1）Env数据结构

Env数据结构代表一个进程描述符，定义在env.h中，包括进程的id，父进程的id，执行状态，该进程的寄存器状态，执行的次数等，并使用env_link指向下一个空闲的Env。

所有Env对象存储在envs数组中，该数组定义在env.c的开头。

除此之外curenv代表当前正在执行的进程，env_free_list指向空闲的进程描述符，组成链表，链表的添加与删除均在表头执行。

（2）进程描述符空间的分配

首先在pmap.c里给数组envs分配空间，然后将分配的页面从free_page_list里剔除掉，最后将UENVS映射到envs，整个过程跟Page数组的处理完全一样，不再赘述。通过检查函数即代表完成。

（3）进程描述符数组初始化

完成env_init函数，将数组中所有的进程id置0，同时将各元素串起来，并把数组地址赋给env_free_list，代码如下：

```
void
env_init(void)
{
    // Set up envs array
    // LAB 3: Your code here.
    int i=0;
    for(i=0;i<=NENV-1;i++)
    {
        envs[i].env_id=0;
        if(i!=NENV-1)
        {
        envs[i].env_link=&envs[i+1];
        }
    }
    env_free_list=envs;
    // Per-CPU part of the initialization
    env_init_percpu();
}
```

（4）初始化进程虚拟地址空间

完成env_setup_vm函数。不同的进程有不同的虚拟地址空间，进而就必须有自己的页目录和页表，该函数的任务就是初始化页目录。

首先分配一个空闲页当做页目录，然后将这个页目录映射内核地址空间（UTOP之上的部分），不需要映射页表因为可以和内核共用页表。

接着增加分配的物理页的引用（JOS设计的一个很不美的地方），将此页的虚拟地址赋值给进程的pgdir，然后使进程有权限操作自己的pgdir。

```
static int
env_setup_vm(struct Env *e)
{
    int i;
    struct Page *p = NULL;
    cprintf("env_setup_vm\r\n");
    // Allocate a page for the page directory
    if (!(p = page_alloc(ALLOC_ZERO)))
        return -E_NO_MEM;
    // LAB 3: Your code here.
    e->env_pgdir=page2kva(p);
    for(i=PDX(UTOP);i<1024;i++)
    {
        e->env_pgdir[i]=kern_pgdir[i];
    }
    p->pp_ref++;
    // UVPT maps the env's own page table read-only.
    // Permissions: kernel R, user R
    e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;

    return 0;
}
```

（5）辅助映射函数

当进行加载二进制代码的时候，需要一个辅助函数region_alloc，其作用是映射虚拟地址va及之后的len字节到进程e的虚拟地址空间里。

实现比较简单，首先将va向下4K对齐，va+len向上4k对齐，以保证分配的是整数页面。之后一个页面一个页面分配即可。

为了使页面用户态可写，需要权限PTE_W|PTE_U。

```
static void
region_alloc(struct Env *e, void *va, size_t len)
{
    // LAB 3: Your code here.
    void* i;
    cprintf("region_alloc %x,%d\r\n",e->env_pgdir,len);
    for(i=ROUNDDOWN(va,PGSIZE);i<ROUNDUP(va+len,PGSIZE);i+=PGSIZE)
    {
        struct Page* p=(struct Page*)page_alloc(1);
        if(p==NULL)
            panic("Memory out!");
        page_insert(e->env_pgdir,p,i,PTE_W|PTE_U);
    }

}
```

（6）进程代码加载

通过load_icode给相应的进程加载可执行代码。可执行代码的格式是elf格式，因此使用类似加载kernel的方式将代码加载到相应内存中。

因为当前JOS没有文件系统，所以用户态的程序是编译在kernel里面的，通过编译器的导出符号来进行访问，通过追踪init.c里的相关代码也可以说明这一点。当然在这里我们无需更多的关注这个可执行代码目前在哪里，只需要完成其加载机制即可。

为了往进程对应的虚拟空间映射到的物理内存中写数据，首先必须要加载进程相应的页目录，当然必须使用此函数外的逻辑来保证在调用此函数之前页目录是已经初始化过的。接着仿照kernel的方式去分析elf文件，加载类型为ELF_PROG_LOAD的段到其要求的虚拟地址中，使用之前完成的辅助函数来方便的进行地址的映射。最后将程序入口放入进程的eip中（后面会有解释），并映射进程堆栈（个人认为堆栈映射应该放在env_setup_vm会更合乎逻辑一些），然后重新加载kern_pgdir，函数返回。

```
static void
load_icode(struct Env *e, uint8_t *binary, size_t size)
{
    // LAB 3: Your code here.
    lcr3(PADDR(e->env_pgdir));
    cprintf("load_icode\r\n");
    struct Elf * ELFHDR=(struct Elf *)binary;
    struct Proghdr *ph, *eph;
    int i;
    if (ELFHDR->e_magic != ELF_MAGIC)
            panic("Not a elf binary");


        ph = (struct Proghdr *) ((uint8_t *) ELFHDR + ELFHDR->e_phoff);
        eph = ph + ELFHDR->e_phnum;
        for (; ph < eph; ph++)
        {
            // p_pa is the load address of this segment (as well
            // as the physical address)
            if(ph->p_type==ELF_PROG_LOAD)
            {
            cprintf("load_prog %d\r\n",ph->p_filesz);
            region_alloc(e,(void*)ph->p_va,ph->p_filesz);
            char* va=(char*)ph->p_va;
            for(i=0;i<ph->p_filesz;i++)
            {

                va[i]=binary[ph->p_offset+i];
            }

            }
        }
        e->env_tf.tf_eip=ELFHDR->e_entry;

    // Now map one page for the program's initial stack
    // at virtual address USTACKTOP - PGSIZE.

    // LAB 3: Your code here.
        struct Page* p=(struct Page*)page_alloc(1);
     if(p==NULL)
         panic("Not enough mem for user stack!");
     page_insert(e->env_pgdir,p,(void*)(USTACKTOP-PGSIZE),PTE_W|PTE_U);
     cprintf("load_icode finish!\r\n");
     lcr3(PADDR(kern_pgdir));
}
```

（7）进程建立

使用上述完成的函数来建立进程。

完成env_create函数，首先分配一个进程描述符，然后加载可执行代码，逻辑很简单。

```
void
env_create(uint8_t *binary, size_t size, enum EnvType type)
{
    // LAB 3: Your code here.

    struct Env* env;

    if(env_alloc(&env,0)==0)
    {
        env->env_type=type;
        load_icode(env, binary,size);
    }

}
```

（8）进程执行

完成env_run函数来运行进程。

首先设置当前进程的一些信息，然后更改当前进程指针指向要运行的进程，之后加载进程页目录跳转到使用env_pop_tf真正使进程执行。

```
void
env_run(struct Env *e)
{

    // LAB 3: Your code here.
    cprintf("Run env!\r\n");
    if(curenv!=NULL)
    {
        if(curenv->env_status==ENV_RUNNING)
        {
            curenv->env_status=ENV_RUNNABLE;
        }
    }
    curenv=e;
    e->env_status=ENV_RUNNING;
    e->env_runs++;
    lcr3(PADDR(e->env_pgdir));
    env_pop_tf(&e->env_tf);
}
```

接着分析env_pop_tf。

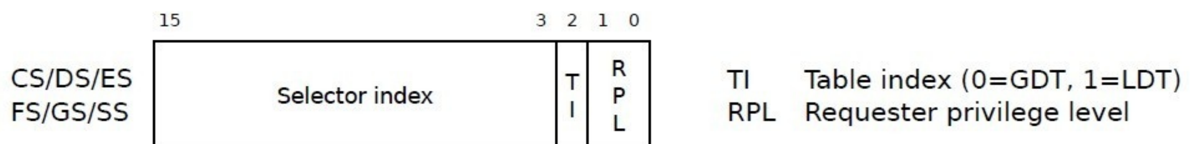
env_pop_tf首先将传入的trapframe，包含所有的寄存器信息压栈，然后使用iret，即中断返回来执行。

```
void
env_pop_tf(struct Trapframe *tf)
{
    __asm __volatile("movl %0,%%esp\n"
        "\tpopal\n"
        "\tpopl %%es\n"
        "\tpopl %%ds\n"
        "\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
        "\tiret"
        : : "g" (tf) : "memory");
    panic("iret failed");  /* mostly to placate the compiler */
}
```

iret到底是什么，这里引用一段IA-32手册上的话：

the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution。

为什么要使用这种方式来运行第一个进程，而不是直接根据该进程入口执行，主要是因为当前运行在内核态（CPL，也就是CS等寄存器的后两位，见图），要使进程运行在用户态必须改变各段寄存器的CPL，但又不能直接给诸如CS等寄存器赋值，所以必须使用iret从堆栈里弹出相应寄存器的值，这也是需要事先将tf的eip放入进程代码入口地址的原因。



做完这些之后，第一个进程就能运行起来了，但立刻又崩溃了，因为没有处理系统调用，而系统调用相关的实现就是下篇日志的内容了。

九、

来源：JOS学习笔记（九）

LAB3代码已经上传。

最近忙于打WOWTCG，早就做完了一直没腾出时间写博客。

LAB3第二部分主要是处理系统调用。

第一部分我们已经让第一个env运行了起来，接着这个env执行一个cprintf，这个cprintf是一个系统调用，因为os暂时没有实现系统调用，所以系统崩溃。

在lab2我们就要完成各种系统调用以及exception和trap等的实现。

handout地址：http://pdos.csail.mit.edu/6.828/2011/lec/x86_idt.pdf

博文中的很多说明都引用了handout里面的图。

一、原理

系统调用、中断以及异常os的处理方式都一样，他们的区别只在于陷入内核的方法不同。

interrupt和exception略有差异，但总体可以认为是某些硬件机制导致我们陷入内核，而系统调用是我们在用户态程序中执行int48进行调用，用户态的调用工作在lib下的syscall.c中。之所以是int 48，是因为T_SYSCALL常量定义为48，理论上定义32--255之间的任何值都可以，0--31规系统使用。详细信息可见handout。

下面详细分析系统调用陷入内核的过程。

首先系统调用的入口在lib下的syscall.c，可以看到，函数syscall使用了嵌入式汇编，先将各个参数分别赋值给eax,ebx,ecx,edx,edi,esi，然后约定将返回值放入eax中（把返回值放入eax的过程是我们需要在内核中实现的），接着使用int 48陷入内核。

剩下的部分中断、异常、陷阱、系统调用都一样（虽然我也分不太清这几个概念），因此下文中除非特殊说明，“中断”一词代表着中断、异常、陷阱、系统调用这4个概念，但唯独中断号和系统调用号进行区分，中断号是指idt表中的索引，系统调用号是指不同的系统调用函数的标识符。

int指令是一个较为复杂的指令，其做了很多事情，按顺序包括以下几步：

1、查找idtr里面的idt地址，根据这个地址找到idt，然后根据idt找到中断向量的表项。

2、检查cpl和表项的dpl，如果cpl>dpl产生保护异常，否则继续

3、根据tssr寄存器里的tss地址找到tss在内存中的位置，读取其中的ss和esp并装载（tss结构是x86定义好的，其内存中存放的位置需要os去决定，并对其中内容的赋值也要os实现，这部分内容在trap.c的trapinitpercpu函数中，同时还包括着加载idt的逻辑）。

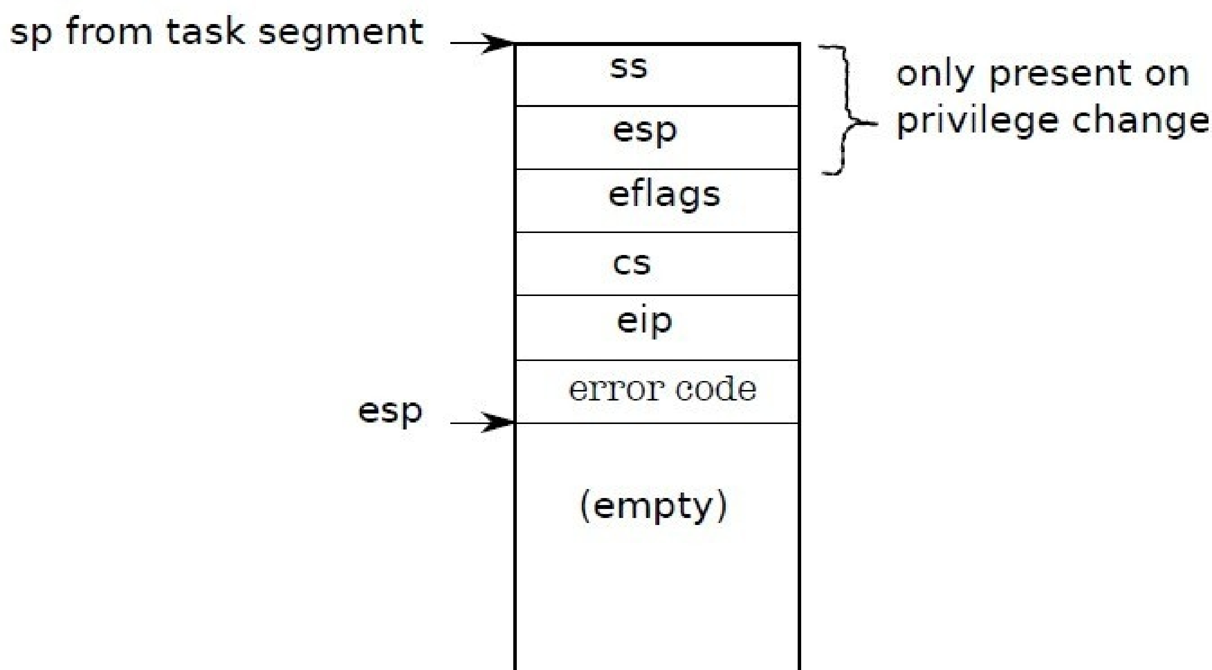
4、如果是一个用户态到内核态的陷入操作，则像堆栈中压入ss和esp，注意这个ss和esp是之前用户态的数据，而不是新装载的数据

5、压入esp,eflags,eip

6、修改eflags中的某些位（比如关中断）

7、如果有必要，压入errorcode，某些中断需要errorcode以及不同中断的errorcode含义可查看handout。

8、根据idt表项设置cs和eip，也就是跳转到处理函数执行。idt内容相关可查看handout。



压入后的堆栈就应该是这个样子的，跳转到相应中断处理函数的时候我们面对的就是这样一个堆栈。接着中断处理函数处理相应操作，然后根据目前堆栈里有的内容和当前寄存器的内容恢复现场，继续程序执行。

二、实验

大概弄懂了原理，接下来解析具体的实验。

1、完成trapentry.S和trap.c的部分内容，使之能正确的调用trap函数，并将一个正确的trapframe结构指针当做函数的参数

在这个实验时，还需要完成很多操作才能进行系统调用。因为加载idt的工作JOS已经帮我们做了，我们需要做的就是初始化idt，给不同的中断分配不同的处理函数。

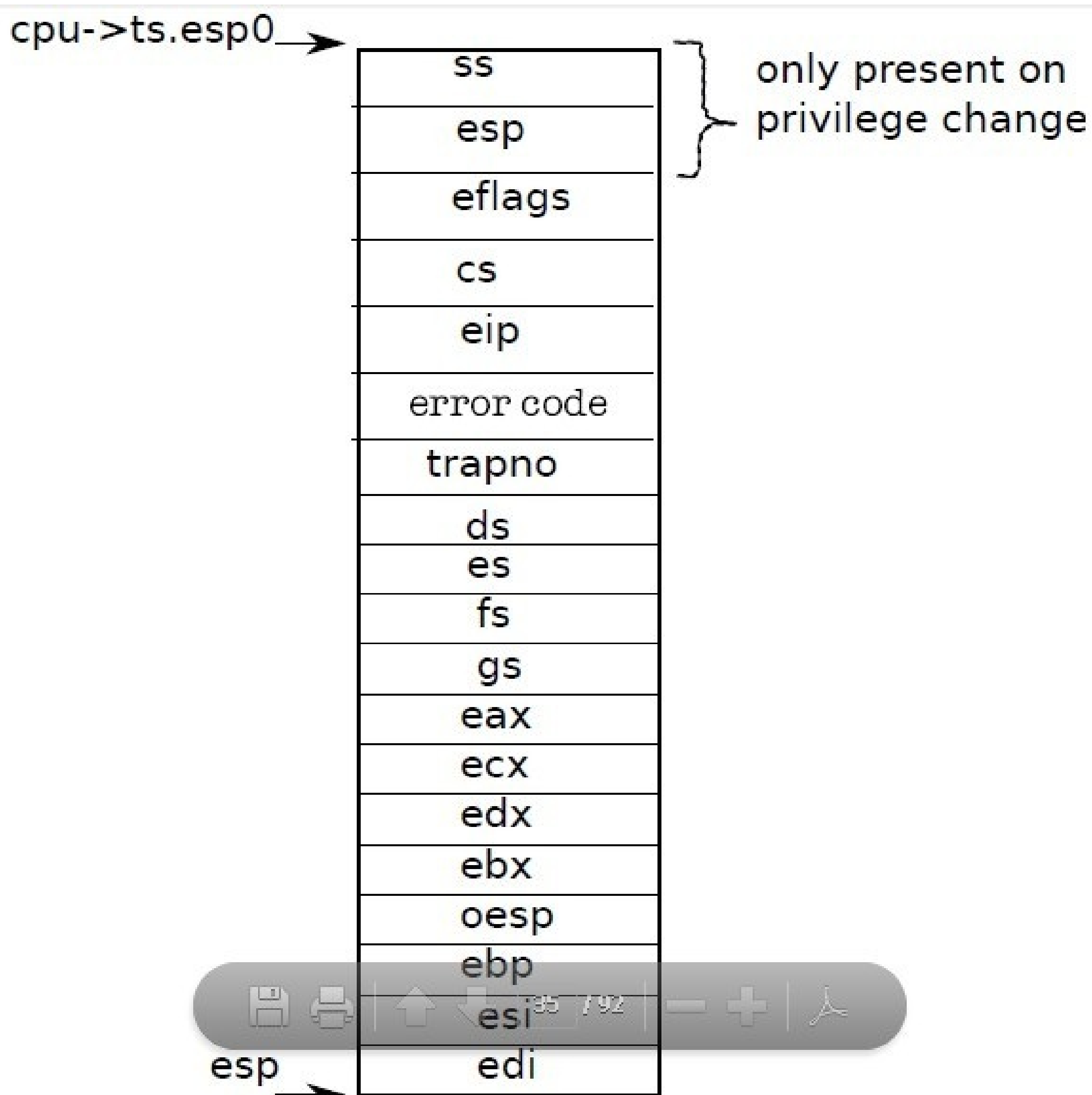
简单分析JOS的逻辑可知，JOS是先将所有中断都跳转到trap函数，再在trap函数里调用trap_dispatch来进行分发，再在trap_dispatch中调用具体的处理函数，虽然这个过程复杂繁琐且个人认为完全没有必要，但至少这意味着idt里指向的函数只需要调用trap函数就可以了。

在trapentry.S中，根据定义的两个宏，参考handout里面errorcode所对应的中断号，可以为不同的中断号定义处理函数，名字随便取，然后再在trap.c中声明这些函数，并获取函数地址填充进idt中（使用setgate宏，使用GD_KT段，也就是OS的代码段），DPL我参考了linux一些信息，将breakpoint,overflow,以及system call设置为3，其余都是0。

此时idt初始化代码就完成了，发现所有中断处理函数跳转到_alltrap处，在trapentry.S中定义_alltrap符号，接着往堆栈里压入某些值，使堆栈看起来像是一个trapframe，此时堆栈栈顶的指针就是指向这个trapframe的指针（结构体内存从低向高增长），将这个指针也就是esp压入堆栈，调用trap函数，进入函数时，会取栈顶元素当做参数，正好就是这个trapframe，这样就完成了我们想要的功能。

接下来考虑如何压栈才能让堆栈看起来像trapframe。

tramframe看起来应该是这个样子的：



在每一个中断处理函数里，已经压入了trapno，所以在_alltrap处，还需要压入trapno以下的所有内容。接着根据要求加载内核数据段，最后再压入esp，执行call trap即可转入c执行。至此exercise 4完成。

2、完成部分trap_dispatch逻辑

这部分没啥可说的，根据中断号把page fault分发到trap.c里的trap_fault_handler函数，然后发现函数里把这个唯一的env给毁掉了，也就是说用户程序出了page fault直接销毁，大概是这么个逻辑。这样exercise 5就完成了。

3、完成breakpoint的中断处理

自己建个函数，然后在trap_dispatch里根据中断号跳转到此函数，在此函数中调用monitor(tf)即可，如果想让到达断点的程序继续执行，可在monitor中多加个命令，退出monitor中的那个死循环，然后trap函数就会根据trapframe里的内容恢复现场继续执行下去。

编程5年总算是知道这个断点是怎么实现的了，囧。

4、完成内核区的系统调用

根据trapframe内的系统调用号（放在eax里）完成相应的系统调用，返回结果要放在eax里，这样经过iret返回时才能得到正确的结果。

5、完成libmain

在libmain用刚才写好的系统调用取得该env的envid，然后根据envid得到Env结构（使用ENVX宏获取UENVS数组的索引），简单的很，也没啥可说的。

6、如果在内核态发生page fault，则panic

在page_fault_handler中判断是否在内核态产生page_fault，判断的方法是查看传入trapframe的cs中的DPL，如果是0，即为内核态。

7、系统调用中的参数检查

在系统调用中有一个是向控制台输出信息的函数sys_cputs，在此函数中要去检查用户传入的字符串所处的内存是否有映射以及是否有read权限，方法也很简单，遍历page table，查看其页表项的PTE_P和PTE_U位即可，如果通不过检测，就把这个envpanic掉。

好了，到此为止应该能通过绝大多数testcase，只有一个testbss除外。

bss里存放着未初始化的变量，和data段不同的是，data段存放变量及其值，而bss段只存放变量及其所占空间大小的信息，所以bss段在elf中所占用的空间要小的多。

现在的问题是，在load_icode时，我们并没有加载bss段。通过objdump来看，按JOS的逻辑，让我们在program header表中只加载ELF_PROG_LOAD类型的段是不会加载bss段的，bss段存放于elf的section表中，且其类型的id是8，因此我们必须写点代码把这个bss段加入内存才行。

但问题是实验中并没有给出这些内容相关的提示，或许是我阅读的材料不够（我没阅读所有相关材料），也许是实验的设计者压根就忘了，反正加载bss段之后就能通过测试了。

代码以上传，因版面问题，就不贴在这里了。

Lab 4

十、

来源：JOS学习笔记（十）

> 神说、进程要有多个、可以分片、切换、互不影响.
>
> 并要支持多任务、多处理器并行. 事就这样成了。
>
> 于是　神造了多个内核栈，又开辟多个寄存器。
>
> 就把这些摆列在内存里、内核空间里、
>
> 管理多核，分别任务.　　神看着是好的.
>
> 有晚上、有早晨、是第四日。

博主写完论文，又写了个简单的编译器，然后回来再拾起来差点烂尾的JOS发现代码已经完全看不懂了，花了一整天时间复习之前自己写的博客，才勉强做了实验四的一点。

本篇博客内容对应课程地址（ http://pdos.csail.mit.edu/6.828/2011/labs/lab4/ ）locking标签之前的内容，也就是多核CPU的初始化工作。博客分以下2部分来进行阐述，1、多核启动流程，2、实验部分

1、多核CPU启动流程

在内核加载和进行初始化时，即便是有一个多核CPU，也只能使用一个核，为了和JOS实验中的描述一致，我们这里暂且把多个核（Core）称之为多个CPU。

内核启动过程中用于执行代码的CPU叫做bootstrap processor (BSP)，其它还未被使用的CPU叫做application processors (APs)，所以在内核执行的时候必然会有的一个过程就是使用BSP来激活AP的过程。

对应JOS代码是init.c中的init主函数，其关键片段如下：

```
    // Lab 2 memory management initialization functions
    mem_init();

    // Lab 3 user environment initialization functions
    env_init();
    trap_init();

    // Lab 4 multiprocessor initialization functions
    mp_init();
    lapic_init();

    // Lab 4 multitasking initialization functions
    pic_init();

    // Acquire the big kernel lock before waking up APs
    // Your code here:

    // Starting non-boot CPUs
    boot_aps();

    // Should always have idle processes at first.
    int i;
    for (i = 0; i < NCPU; i++)
        ENV_CREATE(user_idle, ENV_TYPE_IDLE);
```


lab2的 mem_init，lab3的env_init和trap_init，lab4的mp_init和lapic_init在这之后需要补充一个Big kernel Lock 暂时不用管，然后boot_ap函数启动所有的CPU，接着有多少个CPU就建立多少个ENV。

在启动过程中，mp_init和lapic_init是和硬件以及体系架构紧密相关的，通过读取某个特殊内存地址（当然前提是能读取的到，所以在mem_init中需要修改进行相应映射），来获取CPU的信息，根据这些信息初始化CPU结构，大致流程就是这样，博主因为能力所限，这部分就不进行详细分析了。

boot_ap是个很有趣的函数。在函数中首先找到一段用于启动的汇编代码，该代码和上一章实验一样是嵌入在内核代码段之上的一部分，其中mpentry_start和mpentry_end是编译器导出符号，代表这段代码在内存（虚拟地址）中的起止位置，接着把代码复制到MPENTRY_PADDR处。随后调用lapic_startap来命令特定的AP去执行这段代码。

这段汇编（mpentry.S）中所做的工作和entry.S所做的工作基本相同。需要注意的一点是每个CPU都有自己的寄存器和栈，需要启动的AP目前还处于实模式，并且内存也没有开启分页，因此所有和符号地址相关的操作都要非常谨慎的转换为物理地址。

之后控制流跳转到mp_main，值得注意的是从mpentry.S开始这些代码都是执行在AP上的，此时BSP正在等待（while循环）AP启动成功。在mp_main里可以看到AP初始化自己的env,trap等，接着改变自己的cpu数据结构中的cpu状态标志为启动成功，然后进入死循环空转。BSP得到AP启动成功的信息后接着尝试启动下一个AP。

整个启动流程大概就是如此，下面进入实验过程。

2、实验

2.1

第一个任务是因为我们把代码拷贝到了MPENTRY_PADDR处，所以要在初始化Page的时候把这一页从Page_Free_List中取出来，以防止该页被分配出去导致代码拷贝的失败进而不能正确启动AP。

```
//mpentry_paddr lab4
struct Page* mpentrypage=pa2page((physaddr_t)MPENTRY_PADDR);
mpentrypage->pp_ref=1;
mpentrypage++;
mpentrypage->pp_link=(mpentrypage-2);
```


只需要在pmap.c的page_init函数的最后加上以上几行代码，首先获取该地址对应的Page结构，然后从page_free_list中剥离就行。

如果正确的话，此时应该能通过check_page_free_list函数

2.2

第二个实验要求为每一个核映射其内核栈，也就是对于编号为i的cpu，需要映射KSTACKTOP-i*(KSTKSIZE+KSTKGAP)-KSTKSIZE 到KSTACKTOP-i*(KSTKSIZE+KSTKGAP) 这块虚拟地址空间到符号percpu_kstacks[i]所对应的物理地址处，改变mp_mem_init函数，代码如下：

```
cprintf("map ap stack!");
uintptr_t stacktop;
int i;
cprintf("0x%x 0x%x %d\r\n",bootstack,percpu_kstacks[0],cpunum());
for(i=0;i<NCPU;i++)
{
    stacktop=KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
    boot_map_region(kern_pgdir,stacktop-KSTKSIZE,KSTKSIZE,PADDR(percpu_kstacks[i]),PTE_W);
}
```


值得注意的是，虽然课程中说此时应该可以通过check_kern_pgdir，但实际上是通不过的。原因在于BSP的内核栈被映射了两次，第一次是在lab3中将内核栈映射到了bootstack，当时只有一个CPU，自然映射的是BSP的堆栈；第二次是在我们刚才写的代码里，将BSP（也就是cpunum()==0的CPU）的内核栈映射到了percpu_kstack[0]。诡异的是bootstack地址不等于percpu_kstack地址，而check_kern_pgdir里居然要求两种映射都存在的情况下才能通过。这显然是不可能的，感觉是JOS实验设计中的一个疏忽吧。因此我把检查bootstack的那段代码注释掉了，这样才得以通过。

2.3

第三个实验是需要完成trap_init_percpu()函数，给相应的CPU加载正确的TSS段选择子，代码如下：

```
// LAB 4: Your code here:

// Setup a TSS so that we get the right stack
// when we trap to the kernel.
thiscpu->cpu_ts.ts_esp0=KSTACKTOP - thiscpu->cpu_id * (KSTKSIZE + KSTKGAP);
thiscpu->cpu_ts.ts_ss0=GD_KD;


// Initialize the TSS slot of the gdt.
gdt[(GD_TSS0 >> 3)+cpunum()] = SEG16(STS_T32A, (uint32_t) (&(thiscpu->cpu_ts)),
                sizeof(struct Taskstate), 0);
gdt[(GD_TSS0 >> 3)+cpunum()].sd_s = 0;

// Load the TSS selector (like other segment selectors, the
// bottom three bits are special; we leave them 0)
//cprintf("load code! %d \r\n",cpunum());
ltr(GD_TSS0 +(cpunum()<<3));

// Load the IDT
lidt(&idt_pd);
```


根据函数中的暗示+照葫芦画瓢，不难把代码改成上述的样子。

至此就完成了LAB4的PART A的一小部分，运行目前的OS可以看到（使用make qemu CPUS=4）可以发现创建了9个env，然后出现了未定义的sys_call，系统panic。

十一、

来源：JOS学习笔记（十一）

不知不觉已经写了11篇日志了，本篇博客将完成LAB 4的PART A的剩余部分，包括内核锁、进程（环境）的简单调度算法，以及fork系统调用。

一、内核锁

1、锁实现

考虑到当多个CPU同时陷入内核的场景，若对于关键数据结构不加锁必然就会导致重入错误（如cprintf不加锁会在屏幕上输出奇怪的结果），因此使用锁来保证内核函数内部的逻辑正确性是很有必要的。

内核锁相关代码都在spinlock.c和spinlock.h中，关键代码如下：

```
void
spin_lock(struct spinlock *lk)
{
#ifdef DEBUG_SPINLOCK
	if (holding(lk))
		panic("CPU %d cannot acquire %s: already holding", cpunum(), lk->name);
#endif

	// The xchg is atomic.
	// It also serializes, so that reads after acquire are not
	// reordered before it.
	while (xchg(&lk->locked, 1) != 0)
		asm volatile ("pause");

	// Record info about lock acquisition for debugging.
#ifdef DEBUG_SPINLOCK
	lk->cpu = thiscpu;
	get_caller_pcs(lk->pcs);
#endif
}
```


以及x86.h里面的xchg函数：

```
static inline uint32_t
xchg(volatile uint32_t *addr, uint32_t newval)
{
    uint32_t result;

    // The + in "+m" denotes a read-modify-write operand.
    asm volatile("lock; xchgl %0, %1" :
            "+m" (*addr), "=a" (result) :
            "1" (newval) :
            "cc");
    return result;
}
```


可以看到，xchg函数首先使用lock指令使xchgl变为原子操作，然后尝试将xchgl两个操作数互换，并把原先第一个操作数的结果放入result中返回。

若此时锁是空闲的，则xchg返回0，spin_lock函数执行完成，否则继续执行pause指令，然后接着执行xchg函数直到其返回值为1.

关于lock引用一段汇编手册的资料：

总线加锁前缀“lock”，它是为了在多处理器环境中，保证在当前指令执行期间禁止一切中断。这个前缀仅仅对ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG,DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD,XCHG指令有效，如果将Lock前缀用在其它指令之前，将会引起异常。

关于pause：

提升spin-wait-loop的性能，当执行spin-wait循环的时候，笨死和小强处理器会因为在退出循环的时候检测到memory order violation而导致严重的性能损失，pause指令就相当于提示处理器哥目前处于spin-wait中。在绝大多数情况下，处理器根据这个提示来避免violation，藉此大幅提高性能，由于这个原因，我们建议在spin-wait中加上一个pause指令。（出自于intel 汇编手册）

2、 实验相关：

实验要求在以下4个地方加锁：

（1）i386_init中，启动多个ap之前。

（2）mp_main中，开始把任务调度到cpu上之前。

（3）trap中，若从用户态陷入内核则加锁。

（4）env_run中，从内核态返回用户态需要释放锁。

加锁后，将原有的并行执行过程在关键位置变为串行执行过程，整个启动过程大概如下：

i386_init-->BSP获得锁-->boot_ap-->(BSP建立为每个cpu建立idle任务、建立用户任务，mp_main)--->BSP的sched_yield-->其中的env_run释放锁-->AP1获得锁-->执行sched_yield-->释放锁-->AP2获得锁-->执行sched_yield-->释放锁..... 其中括号表示并行执行

具体代码如下：

（1）i386_init

```
// Acquire the big kernel lock before waking up APs
// Your code here:
lock_kernel();
// Starting non-boot CPUs
boot_aps();
```


（2）mp_main

```
void
mp_main(void)
{
    // We are in high EIP now, safe to switch to kern_pgdir
    cprintf("mp_main!\r\n");
    lcr3(PADDR(kern_pgdir));
    cprintf("SMP: CPU %d starting\n", cpunum());

    lapic_init();
    env_init_percpu();
    trap_init_percpu();
    xchg(&thiscpu->cpu_status, CPU_STARTED); // tell boot_aps() we're up

    // Now that we have finished some basic setup, call sched_yield()
    // to start running processes on this CPU.  But make sure that
    // only one CPU can enter the scheduler at a time!
    //
    // Your code here:
     lock_kernel();
     sched_yield();

    // Remove this after you finish Exercise 4
    //for (;;);
}
```


（3）trap

```
    if ((tf->tf_cs & 3) == 3) {
        // Trapped from user mode.
        // Acquire the big kernel lock before doing any
        // serious kernel work.
        // LAB 4: Your code here.
          if (tf->tf_cs != GD_KT)
            {
                lock_kernel();
            }
        assert(curenv);

        // Garbage collect if current enviroment is a zombie
        if (curenv->env_status == ENV_DYING) {
            env_free(curenv);
            curenv = NULL;
            sched_yield();
        }
```


（4）env_run

```
    if(curenv!=NULL)
    {
        if(curenv->env_status==ENV_RUNNING)
        {
            curenv->env_status=ENV_RUNNABLE;
        }
    }
    curenv=e;
    if(e->env_type!=ENV_TYPE_IDLE)
    {
        //cprintf("envid:%d\r\n",e->env_id);
    }
    e->env_status=ENV_RUNNING;
    e->env_runs++;
    lcr3(PADDR(e->env_pgdir));
    unlock_kernel();
    env_pop_tf(&e->env_tf);
    //panic("env_run not yet implemented");
}
```


二、任务调度

1、原理

在JOS中，任务调度在内核中是函数sched_yield，同时在用户态有相应的系统调用sys_yield也可以调用内核中的这个函数。

i386_init启动时，在boot_ap函数调用后，为每个CPU创建一个idle任务，相应代码在user/idle.c，通过代码可以看到，这些任务使用一个死循环，不断调用sys_yield尝试切换任务。

所以在这种机制下我们需要实现sched_yield函数，具有以下几个要求：

（1）找到状态为runnable的任务，并切换执行

（2）如果找到一个running状态的任务，且此任务执行的CPU为当前CPU，也可将此任务切换执行

（3）若没有runnable任务，则执行idle任务。

（4）从当前CPU执行的任务处开始遍历链表（为了保证公平性）

2、代码

sched_yield的任务调度代码如下：

```
// LAB 4: Your code here
struct Env* thisenv=thiscpu->cpu_env;
int pos=0;
if(thisenv==0)
    thisenv=envs;
   for (i = 0; i < NENV; i++) {
        if (&envs[i]==thisenv)
        {
            pos=i;
            break;
        }
    }
for(i=0;i<=NENV-1;i++)
{
    int j=(i+pos+1)%NENV;
    struct Env* env=&envs[j];
    if((env->env_status==ENV_RUNNABLE  || (env->env_status==ENV_RUNNING
            && env->env_cpunum==cpunum())) && env->env_type!=ENV_TYPE_IDLE)
    {
        env_run(env);
    }
}
```


首先找到CPU当前任务的下标，然后从下标的下一个开始遍历数组。博主这段代码写的比较笨，主要是为了方便调试所用。

同时还要增加系统调用的部分代码，把进入内核态后的系统调用号和具体的系统调用对应起来，较为容易故不再详细论述。

三、fork

1、原理

fork作为一个系统调用，其功能是根据父进程创建出一个一模一样的子进程，若返回的是0则说明是子进程，否则是父进程，同时返回值为子进程的系统调用号。

JOS实现fork过程采用的是用户态"类库"的形式封装了一系列系统调用，包括创建新进程、设置新进程状态、虚拟地址映射等等，这部分已经在user/dumbfork.c中封装好了，实验所要求的是实现相关的系统调用，包括：

- sys_exofork:若为父进程返回子进程号，子进程则返回0。
- sys_env_set_status:设置子进程格式为runnable或者not_runnable
- sys_page_alloc:分配一个物理页并对应到某个虚拟地址
- sys_page_map:拷贝父进程的某个PTE，以此来建立子进程的虚拟内存映射
- sys_page_unmap:解除某个虚拟地址的映射（在PART B中使用）

2、代码

sys_exofork:

```
// LAB 4: Your code here.
cprintf("sys_eexofork\r\n");
struct Env* env;
int result=env_alloc(&env,curenv->env_id);
if(result==0)
{
    memmove(&env->env_tf,&curenv->env_tf,sizeof(env->env_tf));
    env->env_tf.tf_regs.reg_eax=0;
    env->env_status=ENV_NOT_RUNNABLE;
    cprintf("child %d\r\n",env->env_id);
     return env->env_id;
}
else
{
return result;
}
```


可以看出，该函数首先复制各寄存器的状态（env_tf），然后系统调用本身返回子进程的id，因为调用此系统调用的进程为父进程。同时将子进程的eax寄存器设置为0，因为系统调用的结果存放在eax寄存器中，这样子进程返回后得到的系统调用返回结果就为0。

sys_env_set_status

```
cprintf("sys_env_set_status\r\n");
if(status!=ENV_RUNNABLE && status!=ENV_NOT_RUNNABLE)
{
    return -E_INVAL;
}
struct Env* env;
int ret=envid2env(envid,&env,1);
if(ret<0)
    return ret;
// cprintf("set status!!");
env->env_status=status;
return 0;
//panic("sys env set status not implemented");
```


首先判断状态会否合法，然后根据进程ID进行查找，最后设置状态并返回。

sys_page_alloc:

```
// LAB 4: Your code here.
cprintf("sys_page_alloc\r\n");
struct Env* env;
int ret;
ret=envid2env(envid,&env,1);
if(ret<0)
        return ret;
if((int)va>=UTOP)
{       return -E_INVAL;    }
if((perm & PTE_U)==0 || (perm & PTE_P)==0)
{       return -E_INVAL;    }
struct Page* page=page_alloc(ALLOC_ZERO);
if(page==NULL)
{       return -E_NO_MEM;    }
ret=page_insert(env->env_pgdir,page,va,perm);
if(ret!=0)
{
    page_free(page);
    return -E_NO_MEM;
}
return 0;
//panic("sys_page_alloc not implemented");
```


按实验要求判断各种条件。

sys_page_map:

```
cprintf("sys_page_map\r\n");
struct Env* env;
struct Env* dstenv;
    int ret;
    ret=envid2env(srcenvid,&env,1);
    if(ret<0)           return ret;
    ret=envid2env(dstenvid,&dstenv,1);
    if(ret<0)           return ret;
    if((int)srcva>=UTOP || (int)dstva>=UTOP || ((int)srcva & (PGSIZE-1))!=0
            || ((int)dstva & (PGSIZE-1))!=0)
    {           return -E_INVAL;        }
    pte_t* pte;
    struct Page* page=page_lookup(env->env_pgdir,srcva,&pte);
    if(page==NULL)
    {           return -E_INVAL;        }
    if((perm & PTE_U)==0 || (perm & PTE_P)==0)
    {           return -E_INVAL;        }
    if((perm & PTE_W) && !((int)(*pte) & PTE_W))
    {           return -E_INVAL;        }
    ret=page_insert(dstenv->env_pgdir,page,dstva,perm);
    if(ret!=0)
    {           return -E_NO_MEM;       }
    return 0;
//panic("sys_page_map not implemented");
```


主要是一些判断+LAB 2中的函数的封装，没什么可说的。

sys_page_unmap:

```
static int
sys_page_unmap(envid_t envid, void *va)
{
    // Hint: This function is a wrapper around page_remove().

    // LAB 4: Your code here.
    cprintf("sys_page_unmap\r\n");
    struct Env* env;
            int ret;
            ret=envid2env(envid,&env,1);
            if(ret<0)
                    return ret;
            if((int)va>=UTOP ||  ((int)va&(PGSIZE-1))!=0 )
            {
                        return -E_INVAL;
            }
            page_remove(env->env_pgdir,va);
            return 0;
    //panic("sys_page_unmap not implemented");
}
```


到此PART A基本结束了，运行结果如下：

```
roger@roger-virtual-machine: ~/work
4: I am the child!
5: I am the parent!
5: I am the child!
6: I am the parent!
6: I am the child!
7: I am the parent!
7: I am the child!
8: I am the parent!
8: I am the child!
9: I am the parent!
9: I am the child!
[00001008] exiting gracefully
[00001008] free env 00001008
10: I am the child!
11: I am the child!
12: I am the child!
13: I am the child!
14: I am the child!
15: I am the child!
16: I am the child!
17: I am the child!
18: I am the child!
19: I am the child!
[00001009] exiting gracefully
[00001009] free env 00001009
No more runnable environments!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```


PS：写起来怎么感觉这个实验好简单啊，做起来怎么感觉难到爆啊。。。。

十二、

来源：JOS学习笔记（十二）

快找工作了，一直没更新，放假一周的时间抽了点工夫做了LAB4的PART B，总体来说还是感觉比较难的，尤其是一段汇编代码和异常栈那乱七八糟的堆栈。

一、概述

本部分实验主要是实现一个copy on write的fork函数，第一步是实现一个用户态的page fault处理机制：首先用户态使用一个系统调用传递给内核态一个函数指针作为page fault的回调函数，接着当发生page fault时内核进行简单的判断将该函数需要的一个特殊数据结构压栈，再使用iret跳到用户态执行此回调函数，执行完之后接着继续执行原先的用户态函数。第二步是在此基础上实现一个copy on write的fork，首先复制父进程地址空间的映射（也就是页目录和页表），然后把对应的页表全部变成不可写，并在保留位加入特殊符号，因此当写操作时候会报page fault错，报错后转入用户态的，在用户态把出错的页面复制后进行重新映射，之后继续返回源程序执行。

看起来不复杂，实际调试起来非常繁琐，内核crash上百次后总算是调通了。

二、实验

Exercise 7

实现一个设置page_fault_upcall的系统调用，较为简单：

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
    // LAB 4: Your code here.
    struct Env* env;
        int ret=envid2env(envid,&env,1);
        if(ret<0)
            return ret;
    env->env_pgfault_upcall=func;
    cprintf("func :0x%x \r\n",func);
    return 0;
    //panic("sys_env_set_pgfault_upcall not implemented");
}
```

Exercise 8

实现内核态的page_fault处理函数，该函数负责跳转到用户态的upcall（也就是pfentry.S），并为用户态的page_fault_handler设置好参数。

为什么要在用户态进行处理，即使用env_run进而调用而不是直接跳转到upcall函数指针处执行？个人认为，是因为直接在内核态处理过于危险，用户可以借此注入高权限的恶意代码。

```
void
page_fault_handler(struct Trapframe *tf)
{
    uint32_t fault_va;
    fault_va = rcr2();
    if((tf->tf_cs & 3)==0)
    {
        //内核态的错误依然没法处理
        print_trapframe(tf);
        panic("kernel mode page faults!!");
    }
    //判断用户是否给异常栈进行了映射
    user_mem_assert(curenv,(void*)(UXSTACKTOP-PGSIZE),PGSIZE,0);
    if(curenv->env_pgfault_upcall==NULL  )
    {
        //没有注册用户态的upcall函数
    cprintf("[%08x] user fault va %08x ip %08x\n",
        curenv->env_id, fault_va, tf->tf_eip);
    env_destroy(curenv);
    return ;
    }
    //构造数据结构，并复制，这个数据结构将传递给用户态的处理函数
    struct UTrapframe utf;
    memmove((void*)(&utf.utf_regs),(void*)(&(tf->tf_regs)),sizeof(tf->tf_regs));//复制寄存
    (&utf)->utf_eflags=tf->tf_eflags;//复制flags
    (&utf)->utf_eip=tf->tf_eip;//复制eip
    (&utf)->utf_err=tf->tf_err;//复制err
    (&utf)->utf_esp=tf->tf_esp;//复制esp
    (&utf)->utf_fault_va=fault_va;
    int espaddr=0;
    if(tf->tf_esp>=UXSTACKTOP-PGSIZE && tf->tf_esp<=UXSTACKTOP-1)
    {
        //运行到这里说明是在用户态的异常处理函数里产生了异常
        struct Page* page=page_lookup(curenv->env_pgdir,(void*)(tf->tf_esp-4),0);
        if(page==NULL)
        {
            cprintf("non Page ...\r\n");
            page=page_alloc(ALLOC_ZERO);
        page_insert(curenv->env_pgdir,page,(void*)(tf->tf_esp-4),PTE_U|PTE_W);
        }
        memmove((void*)((tf->tf_esp)-4-sizeof(utf)),&utf,sizeof(utf));
        espaddr=tf->tf_esp-4-sizeof(utf);//新的栈顶
    }
    else
    {
    //将UTrapframe放到栈顶
    memmove((void*)(UXSTACKTOP-sizeof(utf)),&utf,sizeof(utf));
    espaddr=UXSTACKTOP-sizeof(utf);//改变栈指针，注意栈的生长是从高到底生长
    }
    struct Env *env=curenv;
    int calladdr=Paddr((int)env->env_pgfault_upcall);
    curenv->env_tf.tf_eip=(int)env->env_pgfault_upcall;//将eip设置为upcall
    curenv->env_tf.tf_esp=espaddr;//设置堆栈地址
    env_run(curenv);//返回用户态执行
}
```

Exercise 9

完成pfentry.S，主要是在用户态的page_fault_handler结束后如何恢复现场并跳回原程序执行。

```
.text
.globl _pgfault_upcall
_pgfault_upcall:
    // Call the C page fault handler.
    pushl %esp          // function argument: pointer to UTF
    movl _pgfault_handler, %eax
    call *%eax
    addl $4, %esp           // pop function argument


   addl $8, %esp
   movl %esp,%eax
   addl $32,%esp
   popl %ebx
   addl $4,%esp
   popl %esp
   pushl %ebx
   movl %eax,%esp
   popal
   addl $4,%esp
   popf
   popl %esp
   subl $4,%esp
   ret
```

这段代码较为难以阅读，首先给出_pgfault_handler结束后的堆栈：

```
// trap-time esp
// trap-time eflags
// trap-time eip
// utf_regs.reg_eax
// ...
// utf_regs.reg_esi
// utf_regs.reg_edi
// utf_err (error code)
// utf_fault_va            <-- %esp
```

然后按顺序汇编代码做了这么以下几件事：

- 首先esp+8，即跳过utf_fault_va和errcode，指向reg_edi。
- 然后把这个esp存放在eax中。
- 接着esp+32，即指向trap-time eip。
- 然后调用popl，此时eip存放在了ebx中，esp指向eflags
- 然后跳过eflags，指向trap-time esp
- 接着把这个esp出栈替代原先的esp。
- 把ebx里的内容，也就是trap-time eip压入新的堆栈里
- 将eax里的内容放入esp，此时esp又重新指向reg_edi
- 使用popal恢复所有寄存器
- esp+4，跳过trap-time eip，然后popf恢复eflags。此时esp指向trap-time esp
- 接着此esp出栈并替换原esp。
- 然后esp-4，即指向我们之前压入的trap-time eip
- 调用ret，弹出指令后堆栈指向trap-time esp所指向的位置，程序能够正常执行。

Exercise 10

完成用户态的set_pgfault_handler函数，较为简单

```
void
set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
{
    int r;
    if (_pgfault_handler == 0) {
        //如果是第一次赋值，则要先非配异常栈，然后再设置upcall
        int envid=sys_getenvid();
        int r=sys_page_alloc(envid,(void*)UXSTACKTOP-PGSIZE,PTE_U|PTE_W|PTE_P);
        if(r<0)
        {
            panic("alloc uxstack fail");
        }
        sys_env_set_pgfault_upcall(envid, (void*) _pgfault_upcall);

    }

    // Save handler pointer for assembly to call.
    _pgfault_handler = handler;
}
```

Exercise 11

首先我发现了一个我在env.c中env_setup_vm中的一个错误，我只复制了页目录，没有复制页表导致所有进程共享了一个页表，一个修改导致其余的也修改。下面是改正后的函数：

```
static int
env_setup_vm(struct Env *e)
{
    int i;
    struct Page *p = NULL;
    cprintf("env_setup_vm\r\n");
    // Allocate a page for the page directory
    if (!(p = page_alloc(ALLOC_ZERO)))
        return -E_NO_MEM;
    e->env_pgdir=page2kva(p);
    for(i=PDX(UTOP);i<1024;i++)
    {
        if(kern_pgdir[i]!=0)
        {

            struct Page* page=page_alloc(ALLOC_ZERO);
            e->env_pgdir[i]=(int)page2pa(page)|PTE_P|PTE_W|PTE_U;
            if(page==NULL)
            {
                return -E_NO_MEM;
            }
            struct Page* kernpage=pa2page(PTE_ADDR(kern_pgdir[i]));
            memmove(page2kva(page),page2kva(kernpage),PGSIZE);
        }

    }
    p->pp_ref++;
    page_insert(e->env_pgdir,p,(void*)UVPT,PTE_P|PTE_U);
    return 0;
}
```

给出fork.c整个文件，较为简单，即使出错也是因为一些粗心导致的错误。

```
// implement fork from user space
```

```
#include <inc/string.h>
#include <inc/lib.h>

// PTE_COW marks copy-on-write page table entries.
// It is one of the bits explicitly allocated to user processes (PTE_AVAIL).
#define PTE_COW     0x800

//
// Custom page fault handler - if faulting page is copy-on-write,
// map in our own private writable copy.
//
static void
pgfault(struct UTrapframe *utf)
{
    void *addr = (void *) utf->utf_fault_va;
    uint32_t err = utf->utf_err;
    int r;
    extern volatile pte_t vpt[];
    if((vpt[PDX(addr)] & (0 |PTE_W |PTE_COW))==0)
    {
        panic("PTE WRONG!!\r\n");
    }

    int envid=sys_getenvid();
    int result=sys_page_alloc(envid,PFTEMP,PTE_U|PTE_W|PTE_P);

    memmove(PFTEMP,ROUNDDOWN(addr,PGSIZE),PGSIZE);
    sys_page_map(envid,(void*) PFTEMP,envid,(void*)ROUNDDOWN(addr,PGSIZE), PTE_U|PTE_W|PT
}

//
// Map our virtual page pn (address pn*PGSIZE) into the target envid
// at the same virtual address.  If the page is writable or copy-on-write,
// the new mapping must be created copy-on-write, and then our mapping must be
// marked copy-on-write as well.  (Exercise: Why do we need to mark ours
// copy-on-write again if it was already copy-on-write at the beginning of
// this function?)
//
// Returns: 0 on success, < 0 on error.
// It is also OK to panic on error.
//
static int
duppage(envid_t envid, unsigned pn)
{
    int r=sys_getenvid();
    int result=0;
    int perm=0;
    if(pn*PGSIZE==UXSTACKTOP-PGSIZE)
        return 0; //整个地址空间除异常栈之外全部进行重新映射
        perm = (perm |PTE_P| PTE_U|PTE_COW );
    result=sys_page_map(r, (void*)(pn*PGSIZE),envid, (void*)(pn*PGSIZE), perm);
    result=sys_page_map(r, (void*)(pn*PGSIZE),r, (void*)(pn*PGSIZE), perm);
    return 0;
}

//
// User-level fork with copy-on-write.
// Set up our page fault handler appropriately.
// Create a child.
// Copy our address space and page fault handler setup to the child.
// Then mark the child as runnable and return.
//
// Returns: child's envid to the parent, 0 to the child, < 0 on error.
// It is also OK to panic on error.
//
// Hint:
//   Use vpd, vpt, and duppage.
//   Remember to fix "thisenv" in the child process.
//   Neither user exception stack should ever be marked copy-on-write,
//   so you must allocate a new page for the child's user exception stack.
//
envid_t
```

```
fork(void)
{
    // LAB 4: Your code here.
    //panic("fork not implemented");
    //cprintf("this is Fork!\r\n");
    set_pgfault_handler(pgfault);
    envid_t envid;
        uint8_t *addr;
        int r;
        extern unsigned char end[];
        envid = sys_exofork();
        if (envid < 0)
            panic("sys_exofork: %e", envid);
        if (envid == 0) {
            thisenv = &envs[ENVX(sys_getenvid())];
            return 0;
        }
    //  cprintf("user : create new env finish! %d\r\n",envid);
        sys_page_alloc(envid,(void*)UXSTACKTOP-PGSIZE,PTE_U|PTE_W|PTE_P);
        extern volatile pte_t vpt[];
        int i,j;
        for(i=0;i<=UTOP/PGSIZE-1;i++)
        {
            if((vpt[i/1024] &(0|PTE_P))!=0 )
            {
                    duppage(envid,i);
            }
        }
        if ((r = sys_env_set_status(envid, ENV_RUNNABLE)) < 0)
            panic("sys_env_set_status: %e", r);
        cprintf("this is Fork finish!!\r\n");
        return envid;

}

// Challenge!
int
sfork(void)
{
    panic("sfork not implemented");
    return -E_INVAL;
}
```